Technische Universität Berlin

FOG COMPUTING PROJECT REPORT

March 17, 2019

Jacek Janczura, Marina Mursa, Clemens Peters,

Nursultan Shabykeev, Ali Karaki, Marius Schidlack, Alexandre Rozier

Table of Contents

List of Figures

List of Listings

Chapter 1

Introduction - Marina

This chapter gives a brief introduction to the content of the project through presenting the motivation behind it, and statement of the goals.

1.1 Motivation

In a world hooked on technologies, a connection to the outside world is crucial, even when a person is in the car. This is the reason behind every car manufacturer trying to come up with some technological ecosystem that won't just connect the car, but also the user to the world around him. BMW technology package called ConnectedDrive has currently over 8 million cars that connect daily to almost 300 micro services that guarantee not only entertainment but most importantly the security of the users. This services range from Map Updates to Real Time Traffic Information to Concierge Services. Due to such a diversified range with high utility for the user, the cars become moving data centers. Aggregating these connections by time, one type of data that results is represented in the form of requests to connect to BMW servers (to get access the the mentioned services) per second (requests/sec)[1]. Having

[1]https://www.bmw-connecteddrive.de/app/index.html#/portal/store

such huge amounts of data generated continuously, a daily pattern can be observed that corresponds to the intuitive rush-hour peaks.

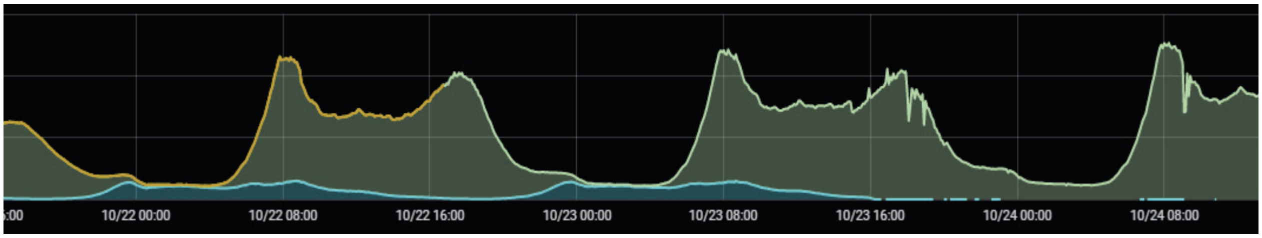


Figure 1.1: Visualisation of BMW data in form of requests per second

1.2 Goals

The primary goal of this project is to develop a framework, based on AWS service, that would be able to detect in a real-time environment a deviation from the pattern mentioned above, label it as an anomaly, and escalate a notification to a tool, e.g. a Slack channel, in the form of an alert accompanied by a small visual representation of the problem.
As a secondary challenge, the project attempts to incorporate a prediction feature, that would deliver information which would allow for predictive scaling of the infrastructure based on a generated short-term prognosis. The purpose of this feature has a deeply financial reasoning behind, allowing the team to adapt the infrastructure besed on the predicted needs, thus reducing the expenses with the AWS services.
The final goal of the project was to come up with an additional set of recommendations and potential improvements to the delivered product. This list of future features that can be investigated by the BMW team in order to assess the prospective value of adding them to their existing framework.

Chapter 2

System architecture - Jacek

We assumed that we might not have an access to the real time data and the only data that will be provided to us might be a batch data. For that reason and to be able to work in parallel the system architecture in our project is composed of 2 approaches: **fixed** and **stream**.

In the fixed approach we will train and test multiple machine learning models, not to wait for whole real time infrastructure. In stream approach we will create the whole streaming infrastructure and than reuse already prepared, trained and tested models from the fixed approach.

At the end we received the following system architecture fig. 2.1. In the final architecture we are using previously gathered data(fixed) to train the ML models. Those trained models will be used for a real time streamed data in which we will try to detect anomalies.

2.0.1 Brief, high level description of the system architecture - Jacek

- **In a fixed approach** we use fixed chunks of data provided by BMW to train the models deployed in AWS SageMaker.
- **In a stream approach** we assumed that the data might come from multiple sources. We gather these data using Kinesis Stream as a queue. Then using a Amazon Simple Notification Service (SNS) lambdas is triggered. This lambdas feed the models with incoming data. In case of anomaly detection feeding lambda is triggering different notification functions using SNS topic to inform user that anomaly has been detected.

A more detailed description is provided in a following chapter.

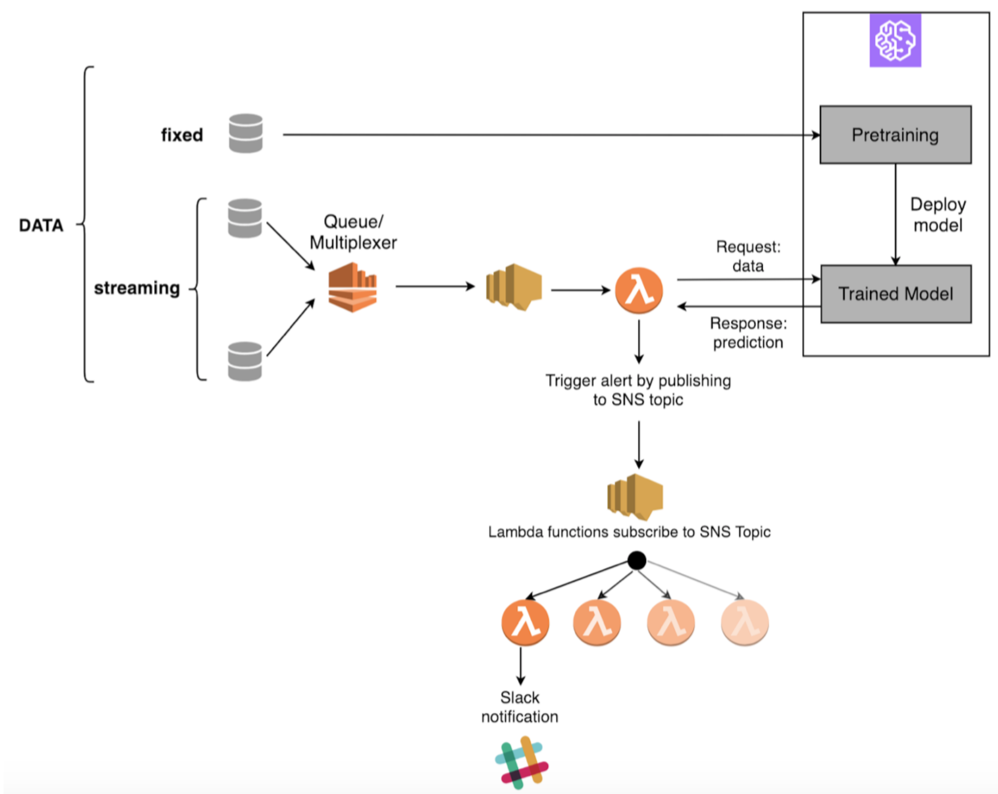


Figure 2.1: System architecture schema

2.1 Data handling

2.1.1 Data generation

When it comes to data in Machine Learning models, in most cases the more training data there is, the better results we can achieve. Unfortunately in case of this project, we did not have the desired amount of data to get the appropriate results from the models. As a short term solution for this problem, the team decided to generate some data that has random anomaly spikes. Below, the two approaches for data generation can be found.

Flowlogs stream generation - Java - Nursultan

Code relative to this section can be found in `utils/flowlogs-stream-generation/`.
Prior we got real data from BMW, we decided to generate the data that slightly reflects patterns of the real data. We needed that generated data to start training and testing our models.
To implement this, we decided to write 2 Java-based multi-threaded services. The first service generates sequential http-requests from EC2 instance named *Data_producer* to the EC2 instance called *Fog_secured_server*. The second service is deployed on *Anomaly_producer* EC2 instance, awakes randomly 2 times per day and sends multiple requests within several minutes, thereby creating an anomaly situation.

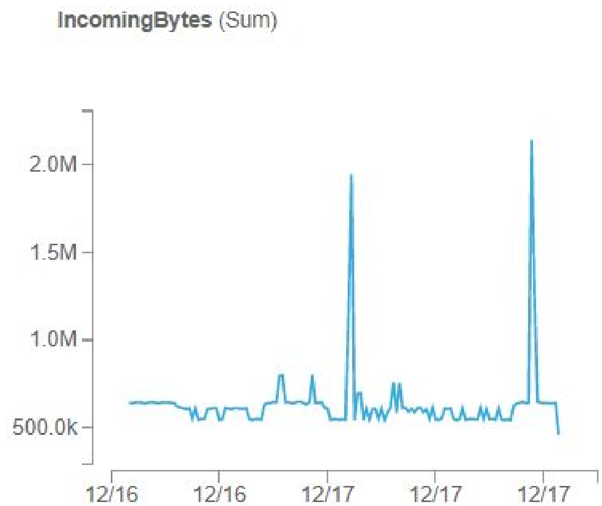


Figure 2.2: The pattern of the generated data

EC2 instances that send HTTP requests are in the same Virtual Private Cloud(VPC)[1]. The EC2 instance that gets requests is within another VPC. There we used VPC Flow Logs feature that lets capture information about the IP traffic that comes in. A flow log record is a space-separated string that has the following format:

```
<version> <account-id> <interface-id> <srcaddr> <dstaddr> <srcport> <dstport> <protocol> <packets> <bytes> <start> <end> <action> <log-status>
```

Figure 2.3: Flow Log record format

AWS provides a set of near real-time data streaming services named Amazon Kinesis Data Streams. We used the service called Kinesis Firehose to load VPC FlowLogs streams. Creation of a Kinesis Data Firehose delivery stream can be done as follows, replacing the placeholder values for RoleARN and BucketARN with the role and S3 bucket ARN where flog logs will be stored:

[1]https://docs.aws.amazon.com/vpc/index.html

```
aws firehose create-delivery-stream \
   --delivery-stream-name 'my-delivery-stream' \
   --s3-destination-configuration \
  '{"RoleARN": "arn:aws:iam::123456789012:role/FirehosetoS3Role", "BucketARN": "arn:aws:s3:::my-bucket"}'
```

Figure 2.4: Creation of Kinesis Data Firehose delivery stream

After the Amazon Kinesis Data Firehose delivery stream is ready and in an active state and the IAM role have been created, the next step to create the CloudWatch Logs subscription filter.

The subscription filter connects Firehose and CloudWatch Logs and starts the flow of real-time log data from the chosen log group to Amazon Kinesis Data Firehose delivery stream.[2]

```
aws logs put-subscription-filter \
    --log-group-name "CloudTrail" \
    --filter-name "Destination" \
    --filter-pattern "{$.userIdentity.type = Root}" \
    --destination-arn "arn:aws:firehose:region:123456789012:deliverystream/my-delivery-stream" \
    --role-arn "arn:aws:iam::123456789012:role/CWLtoKinesisFirehoseRole"
```

Figure 2.5: Creation of subscription filter

After the subscription filter is set up, CloudWatch Logs will direct all the incoming log events that match the filter pattern to Amazon Kinesis Data Firehose delivery stream. It might take a few minutes before the data will be saved into S3. However, it is possible to monitor a size of incoming data and other metrics in a *Monitoring* tab in a Firehose console page.

While creating the subscription filter, it is important to define properly IAM roles and permission policies and correctly associate the permissions policy with the role. The full description of establishing the connection between Kinesis streams and CloudWatch Logs is available in an official documentation.

The data generation pipeline is presented below:

[2]https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/SubscriptionFilters.html

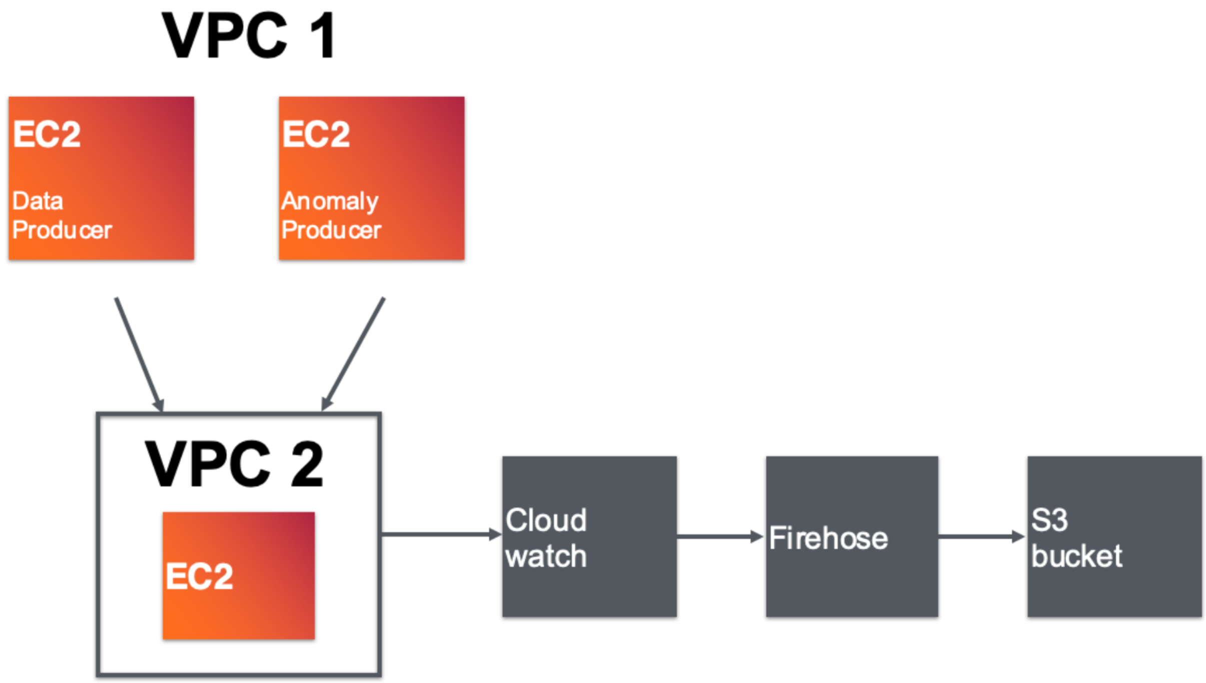


Figure 2.6: Data generation pipeline

Data Generator - Python - Ali

The Dummy Data Generator in Python variant is an alternative version of the one being used currently. It simulates a stream of https requests to a specific IP (EC2 instance inside a VPC). It is highly configurable from scheduled time for requests, has a dynamic server IP for the EC2 instance, and a placeholder for the number of desired requests being sent. We needed an easy to customize script that can generate data fast. This script fulfills this requirement. No complex dependencies and pretty fast executions.
In concept, the workflow to generate dummy data was to do the following:

1. It sends HTTPS requests with our scripts to an EC2 instance
2. The logs from EC2 instance are saved in CloudWatch
3. VPC Flowlogs are saved inside a bucket on S3

4. Flowlogs aggregator lambda function is used to aggregate all the files into actual data set

A request may be rejected or accepted when it reached the EC2, by comparing it to the IP of a machine from the VPC (it can be seen in Fig. 2.7) Requests are sent to a specific IP, the

```python
#Def for a request, stops when the looper is 0
def handle_request(Req):
    #Failed requests for testing
    if (Req.code == 599):
        print('\n',Fore.RED + 'Request Code, FAIL = ' ,Req.code, '\n')
        print(Style.RESET_ALL)
    else:
    #Successful requests for testing
        if (Req.code == 200):
            print('\n',Fore.GREEN + 'Request Code, SUCCESS = ' ,Req.code, '\n')
            print(Style.RESET_ALL)

    print(type(Req), '\n')
    # If you want to print headers, uncomment this
    #print('Request Headers = ',Req.headers)
    global i
    i -= 1
    if i == 0:
        ioloop.IOLoop.instance().stop()

#Requests Looper, you can change the IP of the server down below
def Random_Requester(Number_Of_Requests):
    http_client = httpclient.AsyncHTTPClient()
    global i
    for x in range(Number_Of_Requests):
        i += 1
    #Send data to this IP
        http_client.fetch("http://54.147.181.67", handle_request, method='HEAD')
    ioloop.IOLoop.instance().start()
```

Figure 2.7: Data generation in a python script

requests are categorized into 2. Successful and failed with codes 200 and 599 respectively. The sent data that is already separated into successful and failed requests, and the number of requests are the variables used for anomaly detection.

Anomalies detected with this data correspond to the number of requests recorded in a specific timestamp. So the code logic goes as following:

1. Send failed requests and successful requests to the EC2 instance
2. Randomly send those requests in 2 different methods
3. Every day from Monday to Sunday, randomly send only 1 very abnormal number of requests and 1 very high number of requests

As this script is used only to generate the data the script runs in an infinite loop until it is disabled manually, with a possibility to configure the number of requests and date of sending them.

2.1.2 Fixed data approach - Marius

After working with our fake data streams for some time, we were provided with some real data by BMW. The data was presented in two different formats: Flowlogs and Metrics. They contained network traffic information starting from December 21, 2018. Flowlogs contain raw information about network traffic, e.g. which IP-adress made a request to which server at which point along with other details.

In this section, we will focus on how we used the Metrics files. The Metrics already accumulate the raw network traffic information from the Flowlogs and only give us information about the amount of traffic. More specifically, they contain the amount of successful requests (with HTTP-Code 200) and failed requests (HTTP-Code 4xx and 5xx) at a certain time. Although the files were named *output-mass.json*, they were not correctly encoded in JSON format. An example entry from one of the files can be seen in 2.1.

```
{
    [...]
    'response-code-4xx': {
    [...]
        'Datapoints': [
            {'Timestamp': datetime.datetime(2018, 12, 15, 11, 20,
               tzinfo=tzlocal()),
            'SampleCount': 6.0,
            'Unit': 'None'
            }
        ]
    }
}
```

Listing 2.1: Example entry from metrics data

Some efforts were made to parse the files with a python script. Once all the files are parsed, we take the *SampleCount* values for each entry, and sort them into buckets according to their *Timestamp*. For the time being, we did not differentiate between successful or failed requests, however, this might be a useful thing to do in the future. The buckets were chosen to be of 5 minute granularity in the final product, however we also experimented with 1 minute buckets. A smaller bucket size means higher responsiveness in a real time anomaly detection model, since in order to react to incoming data, we have to wait until the current bucket is full. Furthermore, if the bucket size gets to small, the machine learning model might get vulnerable to noise. Larger buckets on the other side compress the training data for better usability and can make the model more robust. Next, we summed up the value in every bucket to obtain the raw time series, which can be seen in figure ?? in the first picture.

Preprocessing and Cleaning We noticed that the data is still very noisy and contains some impurities. For example, the data between the 26th and the 31st of January was scaled up by a factor of two for unknown reasons. We reduced the noise by applying a smoothing convolution to the data, and scaled down the aforementioned interval (cf. fig. ??). One might argue that in timeseries data, the value of one point in time is highly correlated to the previous ones. Smoothing the timeseries exploits this property to get rid of sudden and unexpected jumps in the training data, or in other words, we reduce outliers and make training of our machine learning models more robust. Finally, we separated one week of data

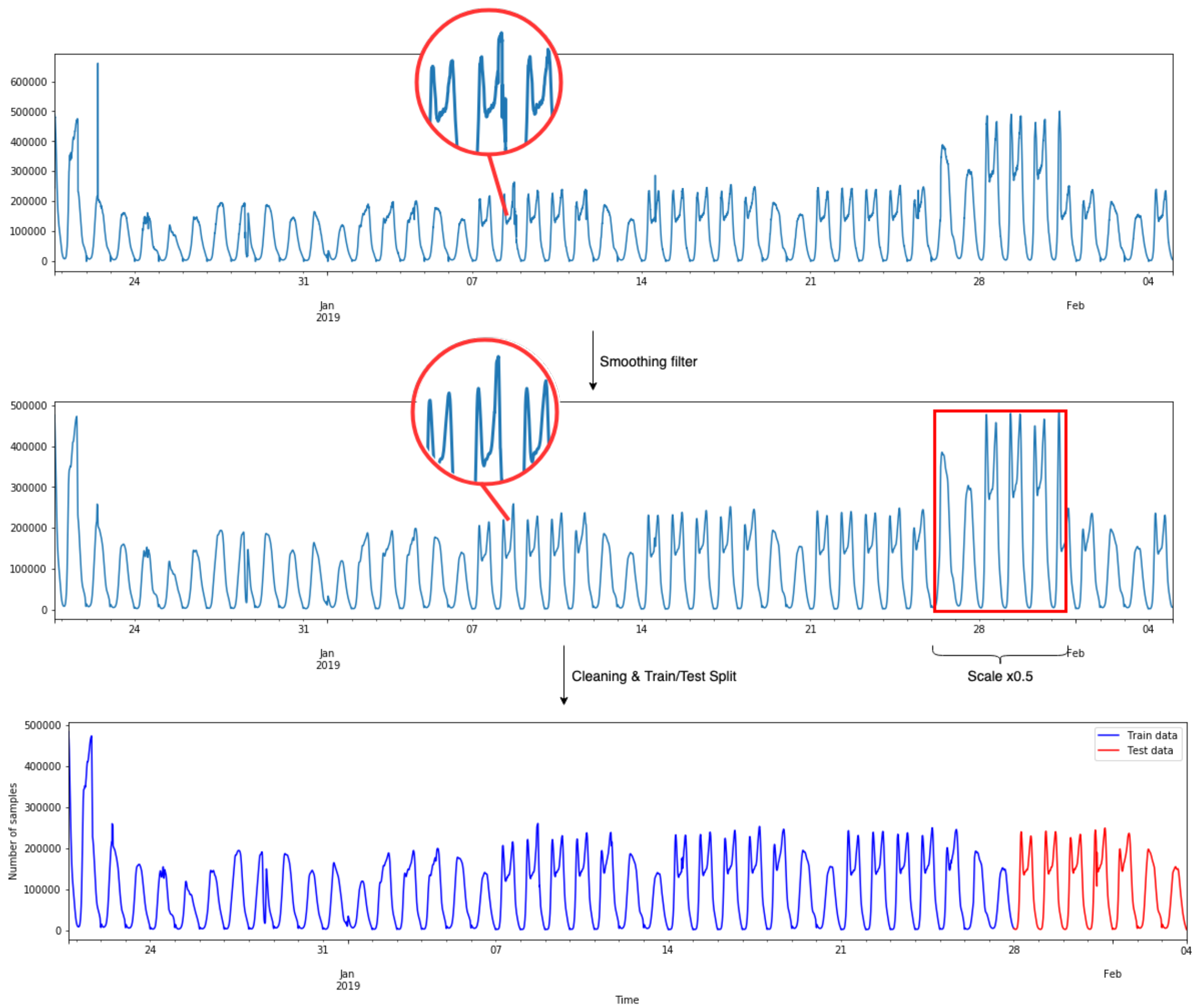


Figure 2.8: Data preprocessing pipeline

for testing purposes.

Exporting for Machine Learning Models For further use with the Machine Learning models described later, the data had to exported in different formats. For the Random Cut Forest and the Mean Predictor, the data can be provided in a simple CSV Format with *Timestamp* and *Value*. The DeepAR Model however requires a more complex format which is described more deeply in chapter 32.

Advantages and Disadvantages While extracting the data from the metrics files is convenient and simple, this approach has several drawbacks: Although we do not have further insight about this, we suspect the metrics files are not generated in realtime, because they accumulate data observed in one day. Thus they can only be used for pretraining a model with fixed data. We have no control over how the metrics files are accumulated. In order to obtain full control over the data stream, one must work with the raw flowlogs files. An approach that utilizes the flowlogs data from scratch is described in section 2.2

2.1.3 Streaming data approach - Alex

Using a custom script hosted on EC2

Code relative to this section can be found in `utils/streaming-data-mock.py`.
Since we were just provided with a fixed dataset and did not have access to the real BMW data stream, we had to create our own mock of it, to test our solution with real-time data. Its principle is quite simple: we extract one default week from the fixed dataset provided by BMW, and use a script to send over and over this particular week on our AWS infrastructure.

An EC2 instance is spun up via Terraform (2.7.4), then provisioned thanks to an Ansible script (installation of the correct python version, required dependencies, upload of the source code and execution).

To be more specific, the script triggers a job every 15 minutes, charged with uploading a new batch of data to a specific entry point on AWS S3. Please note that the data is sent synchronously with the current time, that is, at 8am we will send data concerning the timespan 7:45-8:00am from the default week.

We chose to implement this fake stream in order to test the whole pipeline, because it was the closest we could imitate BMW's infrastructure putting new batches of data to our entry S3 bucket. This was useful during the final presentation (where we could show our pipeline processing live data), and also to work as close a possible to a real-life scenario.

2.2 Real time anomaly detection with Kinesis - Clemens

2.2.1 Motivation

This approach is inspired by a Medium blog post[5]. [3] The goal of the approach is to move the anomaly detection closer to a real-time setup. This means to continuously process the most recent data instead of analyzing data from the previous month or year to find anomalies. To give some more motivation about why this is actually an important issue, here is a quote from a job description from Netflix[4]:

> "Netflix Operational Insight Team is the team responsible for building common infrastructure to collect, transport, aggregate, process and visualize operational metrics. We build powerful systems to allow everyone at Netflix visibility into the state of our environment at both a macro and micro level."

The fact that a leading and modern company like Netflix builds a whole team just dedicated

[3] https://medium.com/@devfire/real-time-anomaly-detection-in-vpc-flow-logs-part-1-introduction-55ed000e039b

to checking and visualizing the current system state in real time, underlines how important it is to know at every moment how your environment performs.
AWS Kinesis is a very suitable tool for this problem for multiple reasons. First of all, Kinesis is designed to process streaming data. This means we can ingest real-time data which is a perfect fit for our given VPC flowlog data. Second Kinesis comes with Kinesis Data Analytics already built in. Amazon Kinesis Data Analytics offers easy ways to analyze streaming data, gain actionable insights, and respond to business needs in real time. In addition to that there is a SQL function called "Random cut forest with explanation" which is described by AWS as follows:

> Computes an anomaly score and explains it for each record in your data stream. The anomaly score for a record indicates how different it is from the trends that have recently been observed for your stream. The function also returns an attribution score for each column in a record, based on how anomalous the data in that column is. For each record, the sum of the attribution scores of all columns is equal to the anomaly score. [1]

2.2.2 Architecture

So with this prior knowledge let's jump right into the system setup. First let's get an overview of the system as it was implemented during the project phase:

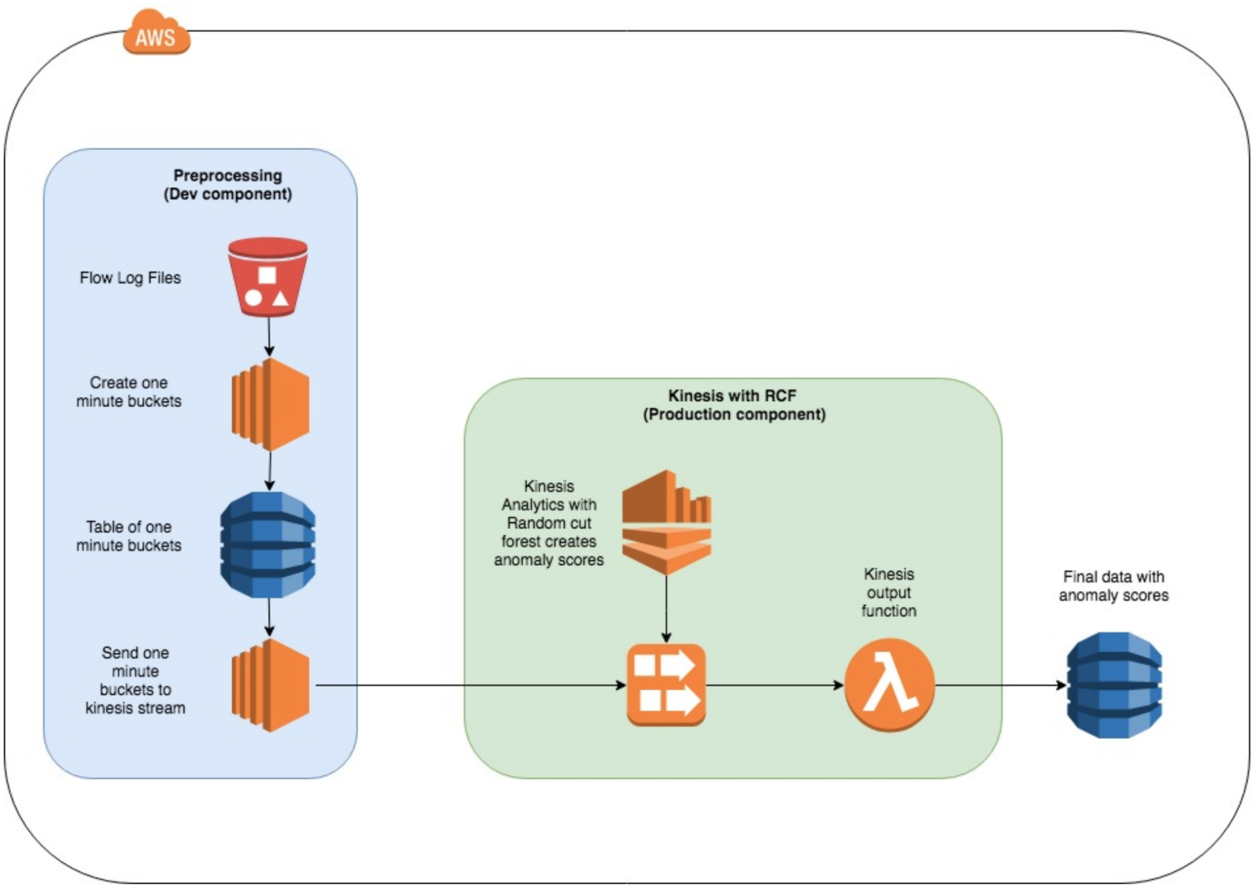


Figure 2.9: Medium Kinesis Random Cut Forest Setup

First of all it should be noted that all the preprocessing components on the left (blue area in figure 2.9) would not be part of a production system. In a production system we would have a continuous stream of VPC flow logs which needs to be handled and preprocessed, whereas in the project setup we had all the flow logs data from the past in a S3 bucket.

2.2.3 Preprocessing components (blue area in figure 2.9)

In the following section we will look at the preprocessing setup in detail, keeping in mind that this is not created to run in production.

S3: Flow Log Files All the flow logs data provided by BMW lies in an S3 bucket called *fog-bigdata-bmw-data*. As mentioned in 10, there are two types of data: first the metrics, which already contain summaries about how many requests were made to the system during a certain time interval. Another issue here was, that the time intervals are of different size (sometimes five minutes, sometimes four or six or something similar). Of course this is not a good basis to run our anomaly detection, because we cannot know what value we expect (since the size of the intervals are different). Therefore the second type of data is more suitable, namely the raw VPC Flowlogs. However, we want to look at how many requests are made to our system per minute and therefore we need to do some preprocessing.

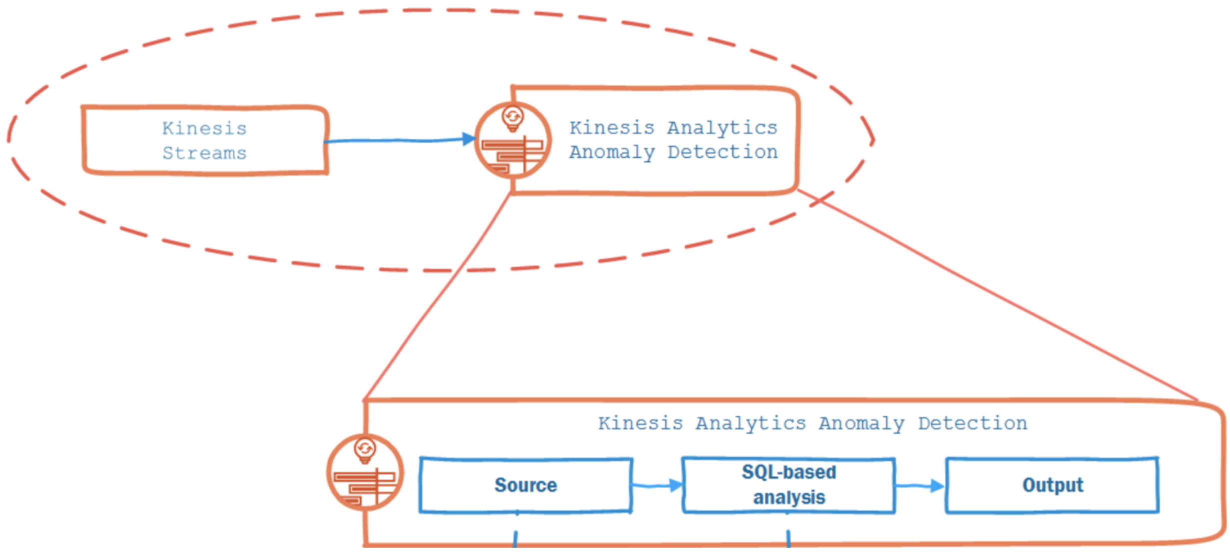
EC2: create one minute buckets This is where the first script with the name *createOneMinuteBuckets.py* comes in. This script is executed on an EC2 instance and reads all the flow log records from the S3 bucket. It creates one-minute buckets from the Flowlogs, so that we know for every one minute time interval how many requests were made to the system.

DynamoDB: table of one minute buckets The information about these one minute buckets is then written into a DynamoDB table (called `medium_bmw_data_to_kinesis`). For the sake of this project the data from the Kinesis table was then exported to a CSV file and the information about the weekdays was added (currently done in Excel). This means that for every one minute bucket we know the hour and the minute and also on which weekday it was recorded. This might be relevant because the expected number of request at 8am on a Sunday can be very different than 8am on a Monday, especially because the requests come from driving cars. Of course this step will not be part of a production system (as mentioned before).

EC2: send one minute buckets to Kinesis stream The next script is called *generateAnomalyScores.py*. The script and the final CSV file is uploaded to EC2 again. The script takes the data from this CSV file and sends it to the kinesis Stream called medium_VPCFlowLogs. An alternative setup could be to read the data directly from DynamoDB and add the information about the weekday in the python script (instead of using CSV and Excel).
This is where the preprocessing part (which would look different in a production system) ends and where the production component (green area in figure 2.9) comes into play.

2.2.4 Production component (green area in figure 2.9)

In this section we assume that we already have all the data we need in the right format. This is the part of the system where the actual anomaly detection happens. This part of the the setup can be used in a production environment in the same or in a similar way. All the preprocessing is done and the data is already fed into the our Kinesis stream.



Kinesis Data Analytics with RCFSource:

https://medium.com/@devfire/real-time-anomaly-detection-in-vpc-flow-logs-part-5-anomaly-detection-d1fc9b61baf8

The Kinesis stream has a Kinesis Data Analytics application which sends the data to random cut forest where the anomaly scores are created. This is the SQL code for the real time analytics:

```sql
1   -- ** Anomaly detection **
2   -- Compute an anomaly score for each record in the source stream using Random Cut Forest
3   -- Creates a temporary stream and defines a schema
4   CREATE OR REPLACE STREAM "TEMP_STREAM" (
5      "dateminute"   BIGINT,
6      "weekdayId"  INT,
7      "numberOfRequests"   INT,
8      "ANOMALY_SCORE"  DOUBLE,
9      "ANOMALY_EXPLANATION"  varchar(512));
10
11  -- Creates an output stream and defines a schema
12  CREATE OR REPLACE STREAM "DESTINATION_SQL_STREAM" (
13     "dateminute"   BIGINT,
14     "weekdayId"  INT,
15     "numberOfRequests"        INT,
16     "ANOMALY_SCORE"  DOUBLE,
17     "ANOMALY_EXPLANATION"  varchar(512));
18
19  -- Compute an anomaly score for each record in the source stream
20  -- using Random Cut Forest
21  -- this is where the anomaly detection happens. See https://docs.aws.amazon.com/kinesisanalytics/latest/sqlref/sqlrf-random-cut-forest-with-explanation.html
22  CREATE OR REPLACE PUMP "STREAM_PUMP" AS INSERT INTO "TEMP_STREAM"
23    SELECT STREAM "dateminute", "weekdayId", "numberOfRequests", "ANOMALY_SCORE", "ANOMALY_EXPLANATION"
24    FROM TABLE(RANDOM_CUT_FOREST_WITH_EXPLANATION(
25      CURSOR(SELECT STREAM "dateminute", "weekdayId", "numberOfRequests" FROM "SOURCE_SQL_STREAM_001"), 100, 256, 100000, 1, true
26    )
27  );
28
29  -- Sort records by descending anomaly score, insert into output stream
30  CREATE OR REPLACE PUMP "OUTPUT_PUMP" AS INSERT INTO "DESTINATION_SQL_STREAM"
31  SELECT STREAM * FROM "TEMP_STREAM"
32  --WHERE ANOMALY_SCORE > 3.0
33  ORDER BY FLOOR("TEMP_STREAM".ROWTIME TO SECOND), ANOMALY_SCORE DESC;
```

SQL code for real time analytics

Source: https://medium.com/@devfire/real-time-anomaly-detection-in-vpc-flow-logs-part-5-anomaly-detection-d1fc9b61baf8

As we can see from the comment in line 32 it would be possible to already define a threshold here. This would then mean, that only anomaly scores which are higher than that specified threshold are sent to the output stream. However it is recommended to do implement this threshold in the output lambda function, to keep the threshold logic separate from the creation of the anomaly scores. Furthermore it is a lot easier to version and backup the lambda function in case the threshold is changed over time than to maintain different versions of the SQL code in the Kinesis Data Analytics part.

Lambda: Kinesis output function The output of the kinesis stream is sent to the Lambda function `medium_bmw_kinesis_to_dynamodb_2`.

This Lambda function just stores the results to another DynamoDB called `kinesis_bmw_anomaly_scores`. Obviously this last database table was just for the purpose of the project phase to collect all the anomaly scores and visualize them in a graph but it is not needed in a production setup (as indicated in the system architecture). Instead in a production system this output function should contain the threshold for the anomaly score and then send a message or trigger a phone call if the threshold is exceeded.

2.2.5 Results

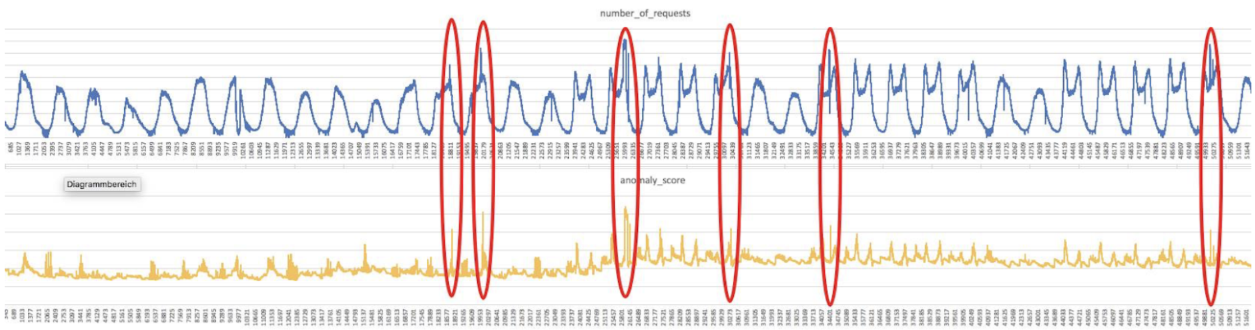


Figure 2.10: Visualisation of Kinesis Analytics Results

The blue graph at the top of figure 2.10 shows the requests to the the system per minute (from the one minute buckets). The orange graph at the bottom of figure 2.10 visualizes the anomaly score at that time. This means if we see an unexpected high peak or unexpected low drop in the first graph (number of requests per minute), in both cases we would expect a peak in the second graph (the anomaly scores). All in all this example does not show a perfect result (since we would expect our anomaly scores to be consistently low when there is no anomaly), however we have to take into account that this particular data set show the requests from December 21, 2018 until January 28, 2019 which means that it starts with the Christmas holidays which does not show the normal pattern of five weekdays followed by two weekends which we see towards the end.

Furthermore this is a series of 54,689 one minute buckets which is not a very long series to train on. Despite these factors of having limited and atypical data the high spikes in the anomaly scores (highlighted by the red ellipses) indicate that the approach in general is quite promising and that it can already detect anomalies.

2.2.6 Future work

The presented results are retrieved from feeding the data into the random cut forest in the following format: date-minute data, the weekday ID (as an integer) and number of requests. To refine the system further it might be interesting to try different combinations of these three input variables to see which combination gives the best results. This way it could be determined if it decreases the performance if we leave out the information about the weekday completely for example.

2.2.7 Implementation guide (Kinesis Random Cut Forest)

This can be found in the appendix of this report (Appendix A) or online at `https://gist.github.com/clemenspeters/8e9025e3bd71e9087df154fb06f96328`

2.3 Anomalies detection algorithms

2.3.1 Amazon SageMaker Random Cut Forest - Nursultan

The code for the instructions below is in GitHub repository:

- *models\random_cut_forest\rcf_bmw_train_deploy.ipynb* - the notebook contains a set of scripts to train and deploy the RCF model
- *models\random_cut_forest\rcf _test_endpoint_model.ipynb* - the notebook contains a set of scripts to test the deployed model

Amazon Sagemaker RCF is an algorithm designed to detect anomalous data points within a dataset. This section describes a creation and a deployment of a SageMaker RCF model.

The data consists of a number of requests (VPC Flowlogs from BMW) aggregated into one minute buckets. To train the model, we used the data over the course of one week that represents a normal pattern. Then, we fed the whole data into the trained model for testing. To start with, it is necessary to specify the locations where we will store our training data and trained model artifacts. In particular, we need the following data:

- bucket - An S3 bucket accessible by an account.
- prefix - The location in the bucket where a notebook's input and output data will be stored. (The default value is sufficient.)

```python
#bucket = 'fog-datasets-west'
bucket = 'kintestingdata'
prefix = 'sagemaker/rcf-benchmarks'
execution_role = sagemaker.get_execution_role()
# check if the bucket exists
try:
    boto3.Session().client('s3').head_bucket(Bucket=bucket)
except botocore.exceptions.ParamValidationError as e:
    print('Hey! You either forgot to specify your S3 bucket'
          ' or you gave your bucket an invalid name!')
except botocore.exceptions.ClientError as e:
    if e.response['Error']['Code'] == '403':
        print("Hey! You don't have permission to access the bucket, {}.".format(bucket))
    elif e.response['Error']['Code'] == '404':
        print("Hey! Your bucket, {}, doesn't exist!".format(bucket))
    else:
        raise
else:
    print('Training input/output will be stored in: s3://{}/{}'.format(bucket, prefix))
```

Figure 2.11: Specifications for data location

Next, we configure a SageMaker training job to train the Random Cut Forest (RCF) algorithm on one minute data.

Hyperparameters

Particular to a SageMaker RCF training job are the following hyperparameters:

- *num_samples_per_tree* - the number of randomly sampled data points sent to each tree.

As a general rule, 1/*num_samples_per_tree* should approximate the estimated ratio of anomalies to normal points in the dataset.

- *num_trees* - the number of trees to create in the forest. Each tree learns a separate model from different samples of data. The full forest model uses the mean predicted anomaly score from each constituent tree.
- *feature_dim* - the dimension of each data point.

Along with these RCF model hyperparameters, we provide additional parameters defining things like the EC2 instance type on which training will run, the S3 bucket containing the data, and the AWS access role. Note that,

- Recommended instance type: ml.m4, ml.c4, or ml.c5
- Current limitation: The RCF algorithm does not take advantage of GPU hardware.[4]

```
from sagemaker import RandomCutForest

session = sagemaker.Session()

# specify general training job information
rcf = RandomCutForest(role=execution_role,
                      train_instance_count=1,
                      train_instance_type='ml.m4.xlarge',
                      data_location='s3://{}/{}/'.format(bucket, prefix),
                      output_path='s3://{}/{}/output'.format(bucket, prefix),
                      num_samples_per_tree=512,
                      num_trees=100)

# automatically upload the training data to S3 and run the training job
rcf.fit(rcf.record_set(num_val_array))
```

Figure 2.12: RCF Model training

[4]https://docs.aws.amazon.com/sagemaker/latest/dg/randomcutforest.html

We used SageMaker Python SDK deploy() function from the job to create an inference endpoint. The function has two input parameters: the instance type and an initial number of instances.

There are two ways to invoke the trained model:

- Right after training in the same notebook, calling a job's *predict(Test_data)* function
- From elsewhere(e.g., inside a lambda function), using a function *runtime.invoke_endpoint (EndpointName, ContentType, Body)*

The model outputs an anomaly score for each input data point. It is suggested in a RCF documentation, to compute a value of the anomaly threshold as 3 standard deviations from the mean score. Any value that is beyond the threshold is considered as an anomaly. The way to compute the threshold value is subject to change. Depending on business requirements, one can calculate it either once for the whole period of data, or for shorter periods of time, e.g., recompute threshold every day.
We decided to experiment and calculate the threshold per each day and these results are available below. The first following plot represents the data, the second one demonstrates testing results. The red plot depicts threshold values that we compute for each day. As we can see, not only high local peaks are detected as anomalies, but also local downfalls and congestions on bottoms are anomaly candidates.

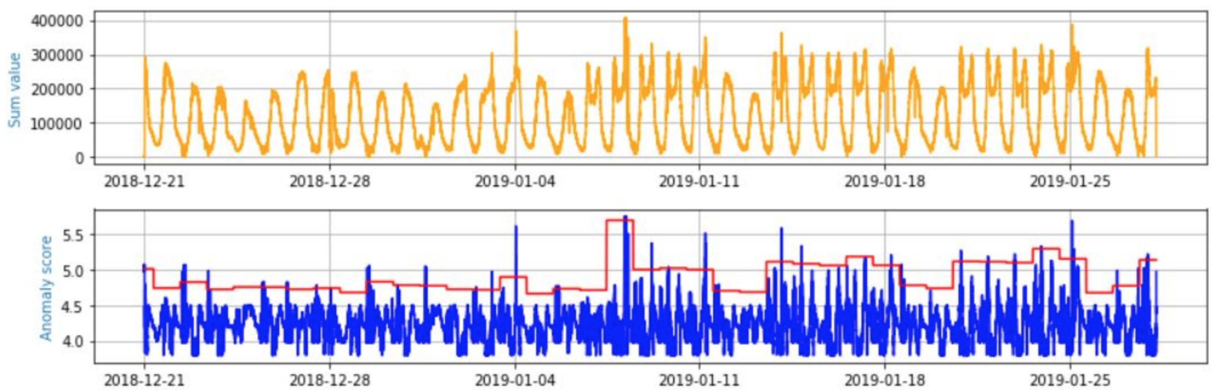


Figure 2.13: Plotted data, anomaly scores, and anomaly threshold values

2.3.2 Mean Predictor Algorithm - Marius

Motivation While sophisticated algorithms like the random cut forest seem to be promising for fitting large amounts of data with complex patterns, they did appear overly complicated for the amount and nature of the data at hand. Apart from holidays, the data shows a clear weekly periodicity, where the data from one week can hardly be distinguished from another. In fact, the weeks are so similar, that we can predict the same pattern every week with high confidence. This motivated us to implement a very simple, yet effective machine learning model called the Mean Predictor.

Idea Unsurprisingly, the name says it all. The objective of the Mean Predictor is to calculate the expected week and standard deviation for each point in a week. Accordingly, we can identify outliers by simply checking if the data point lies within the allowed deviation of the expected value.

Unfortunately, AWS Sagemaker does not offer a Mean Predictor as an Out-Of-The-Box algorithm like the Random Cut Forest. Thus, we implemented it by ourselves.

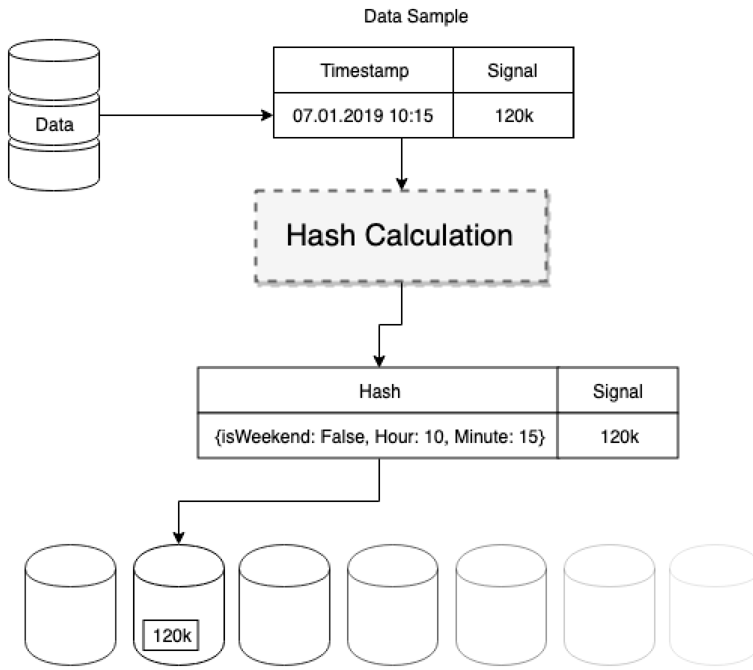


Figure 2.14: Hashing process of Mean Predictor Model

Implementation For calculating the expected week, we sorted every data point into a bucket that is determined by its timestamp. Each bucket has a hash or index associated with them. After distributing all the data into buckets, we calculate the mean of all the values in one bucket, as well as a measure of standard deviation. This process is visualized in figure 2.14.

The hashing function is central to this approach, as we can arbitrarily craft the features that we want our mean predictor to represent. Apart from the weekly periodicity of the data, we also noticed a very predictable daily pattern throughout the working week. Hence we decided to put all working days into one set of buckets, while we put all weekend days into another. We did this to ensure a reasonable amount of samples for each bucket. The hashing function currently employed in the mean predictor is shown in code example 2.2 shows the hashing function employed in our solution. As more and more data comes in, the function can be adjusted to one's needs.

```
def __time_hash(self,t):
        return (t.weekday() < 5,t.hour,t.minute)
```

Listing 2.2: Time stamp hashing function used for Mean Predictor

As mentioned before, we also require a measure of expectation deviation for determining outliers. What first comes to mind is the standard deviation:

$$\sigma_t = \sqrt{\frac{1}{n}\sum_{i=1}^{N}(s_t^{(i)} - \mu_t)^2} \tag{2.1}$$

Where μ_t denotes the mean value of bucket t and $s_t^{(i)}$ the i-th value in this bucket. However, since we square the distance from the mean, the standard deviation weighs outliers much heavier than conform data, which might not be the desired behaviour for the task at hand, because outliers are what we want to detect. Instead one might want to use the average euclidean distance from the mean:

$$\sigma_t = \frac{1}{n}\sum_{i=1}^{N}\sqrt{(s_t^{(i)} - \mu_t)^2} \tag{2.2}$$

In our case, the choice of deviation measure had little influence on the final results, nevertheless if future training data is noisy and contains a lot of outliers, this might be an option to consider.

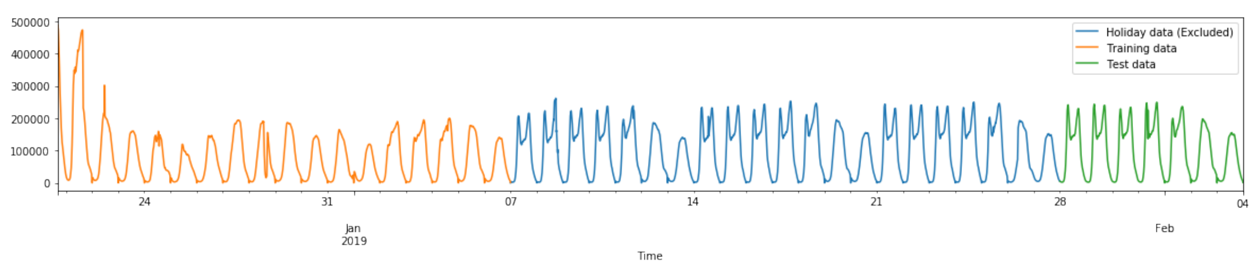


Figure 2.15: Train-Test split of data used for the mean predictor

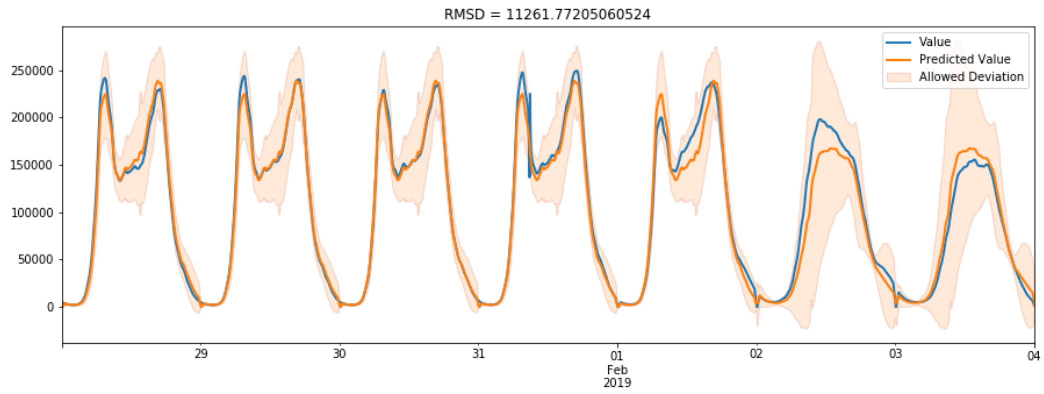


Figure 2.16: One week prediction of mean predictor model

Training and Prediction Due to Christmas holiday, a lot of the provided training data is anomalous, thus not useful for training a model that is supposed to detect anomalies. Accordingly, we excluded this data from training, leaving us with only three weeks of training data. Alternatively one could also alter the hashing function to give holidays a separate set of buckets. Our train-test split is shown in figure 2.15

Once we have expected value and deviation for each bucket, making predictions is trivial. For prediction, the model simply takes a timestamp interval and calculates the hash for each one. We obtain the prediction by simply looking up the expected value and deviation in our table. Figure 2.16 depicts a prediction for the week starting at the 28th of January 2019 and compares it to the ground truth. For this test data, the prediction is highly accurate and the true values never leave the confidence interval. Since we lack real testing data that can be used to assess the outlier detection performance, we must leave it at the intuition that what is outside of the confidence interval is considered anomalous. The RMSE of the prediction lies around 11,261.

Deployment as Sagemaker Endpoint In order to use our model in the Cloud, we had to deploy it as an AWS Sagemaker Endpoint. In the following we describe the technical details related to this task. The endpoint was implemented along the lines of a tutorial provided by AWS [5].

Sagemaker Models are deployed from docker images. Hence, we defined a docker file that encapsulates the model into a container. The container image is then built uploaded to AWS *Elastic Container Registry* (ECR) with the *build_and_push.sh* script. The *train_deploy.py* script provides a convenient commandline interface for spinning up a Sagemaker endpoint using the previously uploaded container image. This script can also be embedded into a terraform pipeline. The model requires only one hyperparameter: The granularity or *frequency* with which it should make the time series predictions. Furthermore it needs a path to the training data located in a S3 Bucket.

At its core, the Sagemaker endpoint is nothing but a REST-API that takes training jobs and prediction requests and forwards it to the Mean Predictor. Once deployed, the endpoint takes prediction requests in the following JSON format:

```
{
    "start": "YYYY-MM-DD HH:MM:00",
    "end": YYYY-MM-DD HH:MM:00
}
```

It will respond with a time series prediction within the *start-end* interval, depending on the aforementioned *frequency*. The response is delivered in the following CSV-Format:

Timestamp	Value	Std
2019-01-05 10:15:00	100	7
2019-01-05 10:20:00	96	13
2019-01-05 10:25:00	112	8
...	...	...

[5] https://github.com/awslabs/amazon-sagemaker-examples/blob/master/advanced_functionality/scikit_bring_your_own/scikit_bring_your_own.ipynb

Similar to the Random Cut Forest, the model will not do outlier detection by itself, but rather provide the user with the necessary information to do so. In order to detect an outlier, one must simply check if a specific data point lies within the bounds of allowed deviation at that time. The user can arbitrarily scale the allowed deviation to their requirements. Such functionality is easily implemented within an AWS Lambda function, which allows seamless integration of the model into the Cloud Architecture. Furthermore, it allows to use the same model for anomaly detection and prediction at the same time.

Conclusion The Mean Predictor is of course a very minimalistic approach: In its current state, we can only incorporate a single source of information, it is not able to leverage additional data, e.g. weather data to improve its prediction. The model is not able to learn features of the time series by itself. As the prediction will resemble a regular working week every time, the model is of little use during holiday periods, however, once enough data is gathered, the hashing function can be manually adjusted to differentiate between holiday and working weeks. With the small amounts of training data we have and the selected test set, the prediction of the Mean Predictor vastly outperforms the other, more complex, Machine Learning models that we examined with an RMSE of just 11,261.

2.4 Prediction algorithms

As presented in the introduction, the team also compared different approaches of time series forecasting for the purpose of predictive auto-scaling of BMW servers. Whenever an amount of network traffic is predicted that lies above or far below of what the servers are able to handle, further resources shall automatically be allocated or released. In this way, the server costs can be minimized, while still ensuring a seamless experience for customers.

2.4.1 Mean Predictor Model - Marius

Naturally, we designed the aforementioned Mean Predictor such that it can also be used for time series forecasting without any additional work required, by simply predicting the expected value for each point in a week.

2.4.2 DeepAR Model - Alex

Presentation

We'll start the presentation of this algorithm by quoting AWS on DeepAR [2]:

> "The Amazon SageMaker DeepAR forecasting algorithm is a supervised learning algorithm for forecasting scalar (one-dimensional) time series using recurrent neural networks (RNN). Classical forecasting methods, such as autoregressive integrated moving average (ARIMA) or exponential smoothing (ETS), fit a single model to each individual time series. They then use that model to extrapolate the time series into the future."

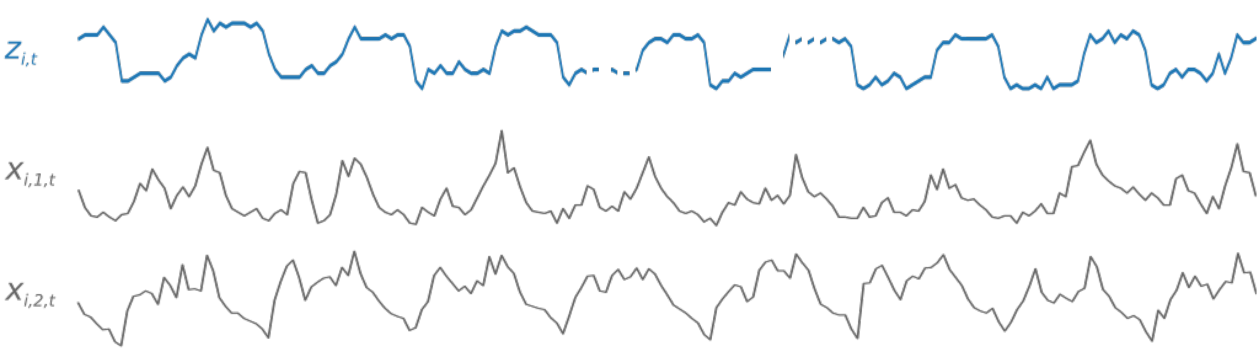


Figure 2.17: DeepAR learns to predict a target (in blue), possibly from dynamic features (in black)

Use-cases

The strength of this algorithm resides in the fact that it is able to learn simultaneously from different correlated time series (called `targets`) and their associated `dynamic features`, which are additionnal time series used as features during training. Please note that it can't be trained on streaming data.

To take a concrete example, let's say we wish to predict the requests hitting the BMW servers on a per-city basis. Our `targets` would be the weekly profile of requests coming from each major city, and the associated `dynamic features` could for instance be the weekly profile of oil prices and temperatures in those cities.

For the sake of clarity, below is an example of what data shape DeepAR expects. Each line corresponds to a `target` time series, and each `dynamic_feat` to a list of its associated dynamic features.

```
{"start": "2009-11-01 00:00:00", "target": [4.3, "NaN", 5.1, ...], "cat": [0, 1], "dynamic_feat": [[1.1, 1.2, 0.5,...]]}
{"start": "2012-01-30 00:00:00", "target": [1.0, -5.0, ...], "cat": [2, 3], "dynamic_feat": [[1.1, 2.05, ...]]}
{"start": "1999-01-30 00:00:00", "target": [2.0, 1.0], "cat": [1, 4], "dynamic_feat": [[1.3, 0.4]]}
```

As a further remark, one can point out that DeepAR supports unknown target values during training, which can be useful if we have an outage at some point cutting access to the data stream.

Behavior on the BMW dataset

We tried three different approaches with the DeepAR algorithm, all having a common agreement: train on all the dataset but the last week, and test on the last week.
It is useful to remark that one could have removed the part of the data being the christmas holidays, but we were curious about the ability of DeepAR to abstract these "outliers". We trained with the following parameters:

1. **Training without dynamic features**

   Idea:

   Training set on all data but the last week, performance evaluation on the last week, without any dynamic features.

   Input shape:

```
1         {"start": "2019-11-01 00:00:00", "target": [4.3, "NaN
             ", 5.1, ...]}
```

   Results:

   Remarkably, the algorithm performs quite well during the working days with a

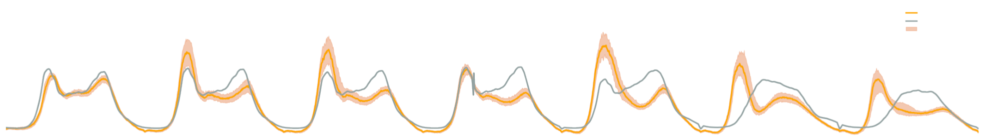


Figure 2.18: DeepAR Prediction trained without dynamic features.

   RMSE (Root Mean Square Error) of 95K. but is unable to learn the trends of weekends. This is quite interesting, since DeepAR is supposed under the hood to automatically encode additional temporal information such as the day of week. Maybe adding some hidden neurons would improve its ability to learn the weekend patterns.

2. **1-Week forecasting training with holidays features**

   Idea:

   Training set on all data but the last week, performance evaluation on the last week, with holidays dynamic features. When a day is a holiday, its associated dynamic feature is a one, else 0.

   We train the model to predict a full week , with a week of context.

Input shape:

```
{"start": "2019-11-01 00:00:00", "target": [4.3, "NaN", 5.1, ...], "dynamic_feat": [[0, 0, 1,...]]}
```

Results:

RMSE (Root Mean Square Deviation) of 72K

3. **1-Week forecasting training with holidays & weekends features**

   Idea:

   Training set on all data but the last week, performance evaluation on the last week, with holidays & weekends dynamic features. When a day is a holiday, its associated dynamic feature is a one, else 0. And similarly, when a day is a weekend, its associated feature is a one, else 0. We decided to add the weekends feature since in the first part we identified that DeepAR had trouble learning their pattern.

   We train the model to predict a full week , with a week of context.

   Input shape:

```
{"start": "2019-11-01 00:00:00", "target": [4.3, "NaN", 5.1, ...], "dynamic_feat": [[0, 0, 1,...],[1, 1, 0, 0, ...]}
```

   Results:

   RMSE (Root Mean Square Deviation) of 122K Adding the weekends dynamic features deeply worsened the model performance, and we think this is because it might conflict with what DeepAR does already perform under the hood. This shows that adding extra dynamic features do not necessary improve this model.

4. **3H forecasting training with holidays & weekends features**

   Idea:

   Training set on all data but the last week, performance evaluation on the last

week, with holidays & weekends dynamic features. When a day is a holiday, its associated dynamic feature is a one, else 0. And similarly, when a day is a weekend, its associated feature is a one, else 0.
The main idea behind this is to have a higher hourly granularity during the forecast, this being useful for instance during predictive auto-scaling.

Input shape:

```
{"start": "2019-11-01 00:00:00", "target": [4.3, "NaN", 5.1, ...], "dynamic_feat": [[0, 0, 1,...],[1, 1, 0, 0, ...]]}
```

Results:

RMSE (Root Mean Square Deviation) of 132K For some reason, this tuning of the algorithm did perform poorly. One of our guess is that the data aggregated in 5-min buckets is too broad, and that we should rather aggregate it in 1-min buckets. Also, it is possible that we are working with too little data as well.

Discussion and recommendations

As mentionned in AWS description, DeepAR outperforms traditional time series forecasting algorithms when the dataset comprises thousands of correlated time series. We tested it on a single time series coupled with only two dynamic features, but it already had promising results (see 2.18), improving continuously with the number of dynamic features.

However, even though DeepAR is supposed to generate additional time features such as day of week, it did not perform well on weekends, and we had to input these features dynamically. This leads us to believe that there is a bit of tweaking to do in order to obtain a foolproof model.

Due to the lack of geolocalized data, we tested this algorithm on a simple task, whereas it is designed to work on a lot more complex task. Thus, we highly recommend that you try it **when additional, geolocalized data will be available, such as traffic per world area. In this case, DeepAR could outperform traditional models and give BMW valuable insight**.

2.4.3 Holt-Winter's Method - Marina

Presentation

An intuitive assumption about the way one can predict the future is that the future should resemble the past experiences. So the most naive approach in forecasting is just to set the position of a future datapoint to the same place as the last observed one. One of common approaches used for forecasting is called *exponential smoothing*. In a simple version we just copy and paste historical data. This however disregards historical data, except for the most recent ones. A more thoughtful approach would be to assign some weight to each data. The more recent a datapoint will be, the bigger its weight value. This is the main idea behind exponential smoothing.

But the simple exponential smoothing can not account for forecasting time series data that has a trend and/or a seasonal component. This issue can be resolved by using an upgraded version of smoothing called Holt-Winters method [3]. It divides a time series Y_t in three parts: one related to its seasonality (s_t), one related to its trend behavior (b_t) and one for the residual part (l_t). For each of them, a simple EWMA is applied to predict a new value. A combination of these expressions is used to estimate Y_{t+1}. The following equations represent the HW computation:

$$
\begin{aligned}
\hat{y}_{t+h|t} &= \ell_t + hb_t + s_{t+h-m(k+1)} \\
\ell_t &= \alpha(y_t - s_{t-m}) + (1-\alpha)(\ell_{t-1} + b_{t-1}) \\
b_t &= \beta^*(\ell_t - \ell_{t-1}) + (1-\beta^*)b_{t-1} \\
s_t &= \gamma(y_t - \ell_{t-1} - b_{t-1}) + (1-\gamma)s_{t-m}
\end{aligned}
$$

Use-case

Going back to the use case of this project, the implementation of this algorithm would theoretically be able to generate a prediction, as well as extract a series of interesting knowledge about historical data.
First of all it would be able to find an existing *trend* in the data. This trend will probably depend on the rate of new cars joining the ConnectedDrive network over time, or on the growing popularity of certain services (that would increase the number of daily request from a car) and it will be best visible after a large amount of data is collected.
Secondly, it should pick up on any *seasonal components*, if they exist, and give a hint towards existing outside factors that can influence the traffic. A hypothesis can be the fact that in summer, when kids do not have school and lots of people chose to go on vacations, the number of cars on the streets is smaller compared to winter. Once again, having enough data (for at least a year), these type of hypothesis can be tested by training the model and decomposing the data.

Currently, the amount of real data that was put at disposal for this project was enough to train a model, but not exactly enough to identify hidden knowledge. The data was simply too little to pick up any significant information, except for the evident ones like the fact that the data has a weekly and a daily pattern.

To see if such an algorithm is applicable to a dataset, first of all we can run a decomposition function that would try to split a batch of data into 3. In figure 2.19 (Part.1) you can see the results from decomposing the BMW dataset. If we analyse the graphs, we can observe that the model is doing a fairly good job at handling the data. We purposely included the anomalous first two days (21-22 December) into our training data to observe how well does the model pick up on the existing trend as well as how easy it is to through it off. We can observe that the trends are well preserved, although their specifics are dimmed down.

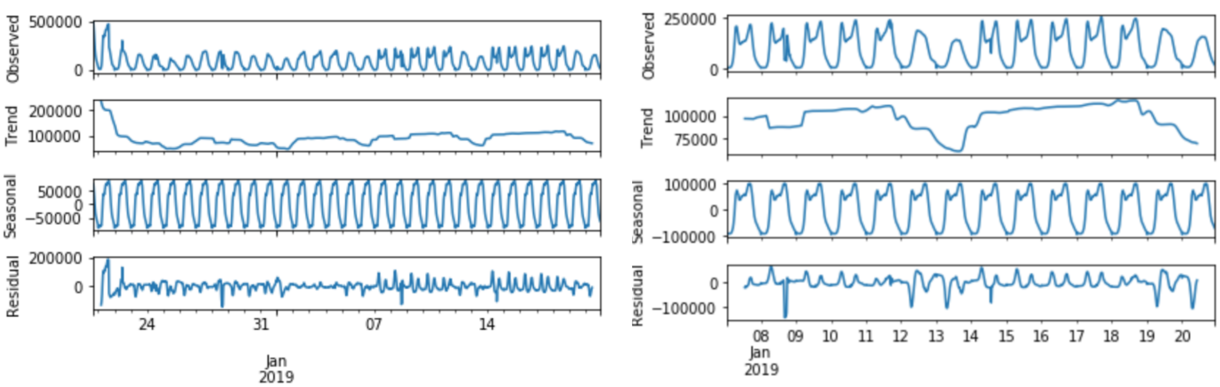


Figure 2.19: Data decomposition into Trend, Seasonality and Residual using Holt-Winter Method

In comparison, the seasonal graph does not do so well, while it does detect a daily pattern, it only plots almost gaussian distribution of daily data. This is due to the poor quality of data, that includes a holiday period, where the rush hour pattern is non-existent. Another factor that diminishes the pattern is the fact that the model averages over weekends, where again the double-peaked pattern disappears. If we exclude this holiday period, we can see a significant improvement in the pattern (see fig. 2.19 Part 2).

Now that we know that our data is suitable for a Holt-Winter's Method analysis, we run divided the data into training and test data.

```
model = ExponentialSmoothing(train, seasonal_periods=seasonl,
    seasonal='mul').fit()
pred = model.predict(start=test.index[0], end=test.index[-1])
```

Listing 2.3: Exponential Smoothing method

The *statsmodel* module in Python has a pre-built function to do the so-called **Triple Exponential Smoothing**. In this method, we can implement both the *additive* and *multiplicative* technique. The additive method is preferred when the seasonal variations are roughly constant through the series, while the multiplicative method is preferred when the seasonal variations are changing proportional to the level of the series. The constants α, β, γ (that were mentioned in the formulas above) - can be estimated (usually through a trial and error process known as fitting) and can be left for the algorithm to estimate or manually set in the .fit()[6] method as following:

```
ExponentialSmoothing.fit(
    smoothing_level=None, smoothing_slope=None,
    smoothing_seasonal=None, damping_slope=None,
    optimized=True, use_boxcox=False,
    remove_bias=False, use_basinhopping=False,
    start_params=None, initial_level=None,
    initial_slope=None, use_brute=True
)
```

Results

Bellow you can see the results of running such an algorithm. Although the **RSME** is about **48K**, which is a relatively good performance and as seen in fig. 2.20 the prediction results are close to the reality, they still have the problem of recreating the double peaks pattern as it has no means of separating the weekdays from weekends, and end up averaging everything together. In addition, because the model puts together three distributions (trend, seasonality and residual) every time it trains, such an algorithm requires a lot of computational power.

[6] https://www.statsmodels.org/dev/generated/statsmodels.tsa.holtwinters.ExponentialSmoothing.fit.html#statsmodels.tsa.holtwinters.ExponentialSmoothing.fit

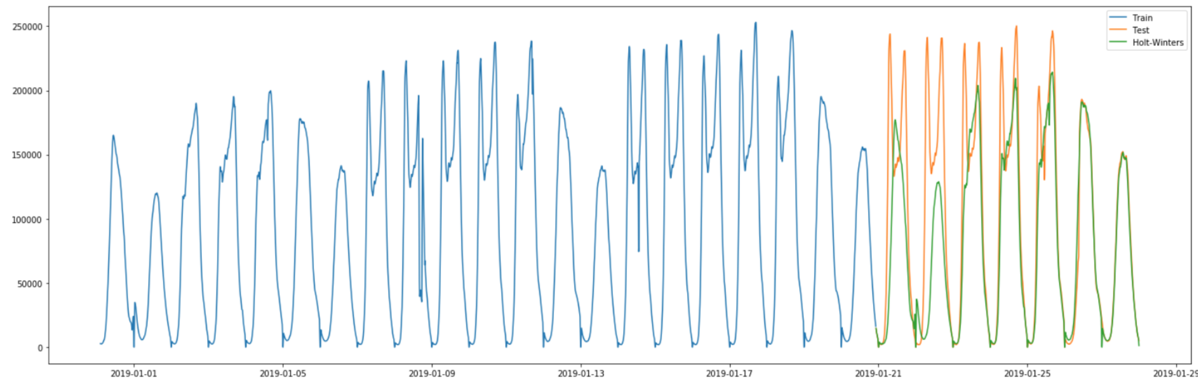


Figure 2.20: Results of Holt-Winter's prediction model

This being said, with a reasonable amount of data for training, for at least a full year, this model should perform significantly better, being able to detect deeper knowledge, beyond the known weekly and daily pattern.

2.5 Data Management - Ali Jacek

In the following chapter different ways of connecting the models with the data are described, as well as what was used to create a work-flow of Data-Model-Results in Notifications.

2.5.1 Mean Predictor V1

In this chapter the first prototype of mean predictor will be described. In the pipeline to detect anomalies: lambdas, models, and S3 buckets, to create the work-flow of sending data to a model and retrieving back results, were combined. We had a lot of issues working with lambdas, layers, dependencies and overall usage of a single lambda across several users.

A great challenge was discovering usage of layers. Tool that enables easy possibility of importing several additional python libraries to a lambdas. Thanks to the layers, lambdas development process can be accelerated and shared among other developers. AWS built in libraries no longer limit us and lambdas code can be viewed in the AWS console. Layers even though they are very powerful are badly documented and it took us a great deal of time to discover the correct usage.

Mean Predictor was a simple model we first created, and the lambdas used to make it fully functional revolved around accessing an S3 bucket with the correct permissions and policies, retrieving data or modifying data in said bucket, then sending it to the model. Input data format is json and the results where either json or in CSV format, according to our needs we changed the formats on demand.

Two main functions where used here, `Model\_Data\_Join.p` and `AnomalyDetection.py`. We also used a function `alertDemo.js` for slack integration and alerts. Our first phase of testing how can we connect all of these together took a while, but we managed to make it work. Each function of those used different policies and dependencies, and AWS has some limitations with uploading .zip files larger than a specific size. We also created Layers, acting like a placeholder for all dependencies that are being used. In addition to those, we also made an API Gate-Away named `RCF\_Data\_Join\_SageMaker` that gives us the possibility to call the model and send some data to it, without actually being in AWS console at all. We used PostMan for the requests. Our plans where to create simple web-page that you can upload a data file to it, and gives back the result instantly.

Model_Data_Join.py can be found in the `/lambdas` folder in the github repository. This function main job is sending data to the model. Using a specific prefix directory, we let the function access our data bucket and look up all keys inside it. The sent data is going to be tested for anomalies, so we couldn't just send a VPC-FlowLog to the model. Instead, we created a file reader, where we read all the keys inside the directory (Keys are FlowLogs), we parse a small json file with index + number of lines inside them. Number of lines = number of request inside each single Flowlog. This was the most convenient way to send requests number to the model using FlowLogs (Fig. 2.22).

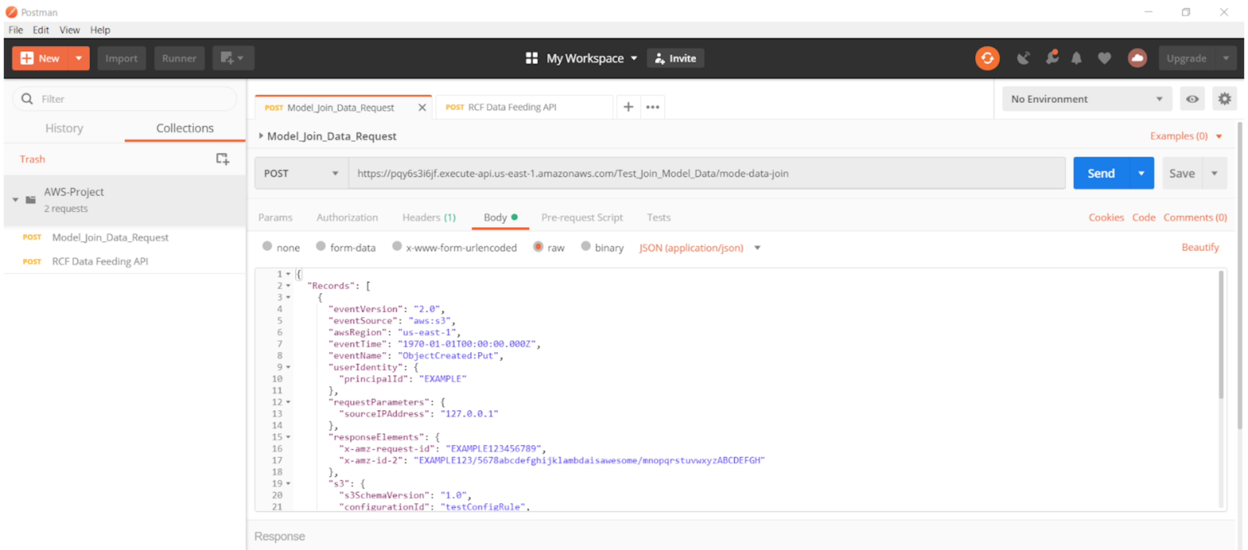


Figure 2.21: Example of API Gateway using Postman

After the data has been parsed from the keys, we get a list, converted to a JSON file later dumped inside the bucket to keep as a history of data that has been used. This process also involves sending the data to the model, after parsing it to json format. We used the invoke method provided from AWS to achieve this. The method takes another lambda function `AnomalyDetection` and the data file as a variable `(json\_data)`. Below you can see the code for this.

After sending the data to `anomalyDetection` lambda, we parse the event back to CSV format for the model to understand it. We separate each value by a new line. Result from the model is then parsed and passed through a prediction method to give us which values where TRUE as anomalies. This result is sent to our AWS channel using an SNS topic (arn:aws:sns:us-east-1:746022503515:api-test) created to queue all notifications. Since this was our first demo, we didn't enable automatic triggering with every new data file, instead, we created test events with the needed buckets and triggered them manually. To make it work properly, you just need Flow-Logs inside the directory (as shown in Fig. 2.24).

```python
def lambda_handler(event, context):

    #Grab the directory with all needed keys (Key is a file)
    list_keys('fog-bigdata-logs', '/streamed2018/12/14/')
    i = 1
    j = 1
    print('Number of Files in this directory = ', len(keys_list))

    #Loop to read each key
    for x in keys_list:
        #Connect to S3 and get File name then read it
        file_obj = event["Records"][0]
        print("\n#########\n\n",i,"- Filename: ", x)
        #Grab a single key and read it's content
        fileObj = s3.get_object(Bucket = "fog-bigdata-logs", Key='/streamed2018/12/14/' + x)
        file_content = fileObj["Body"].read().decode('utf-8')
        #Number of requests is the number of lines in this file
        requests = len(file_content.split('\n'))
        print("Number of requests = ",requests)
        if i<32:
            json_list.append(tuple((i, requests)))
        i += 1
        # Uncomment this to get file content
        # print(file_content)

    #Json dump
    json_result = dump_to_json(json_list)
    # return dump_to_json(json_list)
    return json.loads(json_result)

#Function to list keys in a specific bucket according to a prefix
def list_keys(bucket_name, prefix):
    for key in s3.list_objects(Bucket=bucket_name, Prefix=prefix)['Contents']:
        if(key['Key']).endswith('.bucket'):
            keys_list.append(key['Key'].split("/")[-1])
```

Figure 2.22: Lambda Key Reader

2.5.2 RCF (Fixed Data)

RCF Random Cut Forest is unsupervised algorithm for detecting anomalous data points within a dataset and was used in a SageMaker as our second trial model.

The data connection with this model was a bit more complex; even though the model took

```python
#Function responsible for sending a list of data, converted to json, as a request to the other function that calls the model
def dump_to_json(list):

    #change region_name as per the destination lambda function region
    invokeLam = boto3.client("lambda", region_name="us-east-1")

    data = {}
    data["data"] = list
    json_data = json.dumps(data)

    #Write to json file

    file_name = 'json_dumps_{}.json'.format(timestr)
    # file_name = 'json_dumps.json'
    lambda_path = "/tmp/" + file_name
    s3_path = "json_dumps/" + file_name
    s3 = boto3.resource("s3")
    s3.Bucket('fog-bigdata-logs').put_object(Key=s3_path, Body=json_data)

    #This will invoke @Marius function, Anomaly Detection.
    resp = invokeLam.invoke(FunctionName = "anomalyDetection", InvocationType = "RequestResponse", Payload = json_data)
    # print(resp["Payload"].read().decode())

    print (json_data)
    return(resp["Payload"].read().decode())
```

Figure 2.23: JSON Dump code

normal json format, we had to separate the values and parse the results several times to make a clean execution. The model takes a list of data points, then returns back a result which is `Scores\_Array`. These scores have to be calculated again using a specific function to get the anomalies (above or below the RCF threshold). Based on those values, we trigger an alert that sends notifications to our channels. However, the procedure also was changed to almost fully automatic! We added a trigger to the functions `rcf-sagemaker-testdata` with a suffix (.csv) and objectCreated event over an S3 bucket, which triggers the whole work-flow by itself. An API gateway `RCF\_API` was also created for the same purpose of our first model.

Two S3 buckets were involved in this process, `rcf-sagemaker-finished` and `rcf-sagemaker-testdata`. A small overview of the work-flow is described below:

1. Make sure the model endpoint is working properly (preliminary checks per usual)
2. Upload data to 1 bucket `rcf-sagemaker-testdata`
3. This upload to S3 bucket automatically triggers using SNS topic all subscribed lambda

```python
def lambda_handler(event, context):
    print("Received event: " + json.dumps(event, indent=2))

    #Receive the event (data from first lambda function) and convert to json readable format
    data = json.loads(json.dumps(event))
    payload = data['data']
    # Convert the data to csv format
    payload = list(map(lambda d: str.join(',',map(str,d)), payload))
    payload = str.join('\n',payload)

    #call the model
    response = runtime.invoke_endpoint(EndpointName=ENDPOINT_NAME,
                                       ContentType='text/csv',
                                       Body=payload)

    # Parse the csv response from the model, which is the result
    res_str = response['Body'].read().decode("utf-8")

    #Run predictions calculations
    predictions = list(map(lambda st: st == 'True',str.split(res_str,'\n')[:-1]))

    #Assign it to SNS topic for notifications publishing
    sns = boto3.resource('sns')
    topic = sns.Topic('arn:aws:sns:us-east-1:746022503515:api-test')

    response = topic.publish(
        # TargetArn='arn:aws:sns:us-east-1:746022503515:api-test',
        Message=json.dumps(dict(
            default=str(predictions)
        )),
        Subject='alert',
        MessageStructure='json',
    )

    return predictions
```

Figure 2.24: Lambda precision

functions.

4. 2 functions are then triggered. The data from S3 bucket is read, prepared and send to the model. Since the model is already feed with a new data, the input data from the initial S3 bucket are moved to the finished sets to `rcf-sagemaker-finished` S3 bucket.

5. Results and especially the anomaly detection, trigger the notification functions which inform the user about the anomalies.

Used functions The whole process involved 2 functions, `RCF\_Data\_Join\_SageMaker.py` and `RCF\_Anomaly\_Model\_2.py`. The first function is triggered by file upload to the S3 bucket and reads uploaded data. After parsing the data the function sends the filtered data to the second function, where we call the model for analysis. All policies, notes, used variables, endpoint names etc. can be found in github repository under `Notes.txt`. The function reads all files in the S3 bucket and lists them for logging purposes. Once we upload a file, it separates it into 2 versions. The data is usually DATE SCORE format. So we create 2 lists, 1 with both values and 1 that has only the scores data. We do this because we want to preserve the date as time-stamp history for the model. Once we have the parsed data, the process of sending this data starts. The smart thing about this is that we used AWS functions to clear our buckets. The data gets sent from `rcf-sagemaker-testdata` bucket to `rcf-sagemaker-finished` bucket, with the name changed to include time of the data testing and date as shown in the Fig. 2.25.

We then proceed to call `dump\_to\_csv`, where we send the data to the model including the date time-stamp, and also create a `csv\_dump` folder inside the finished bucket for testing and debugging purposes.

The model function, `RCF\_Anomly\_Model\_2.py` is then invoked with the needed json data. And as a result of some time consuming calculations in this function, we had to make the run-time of lambda up to 3 minutes to avoid any unwanted failures.
After the json event is received, we parse it back to the correct csv format and send it to the model using `invoke\_endpoint` method. The result is a huge CSV response with all the values we need to calculate with RCF for anomalies.
The final cut-off score is calculated with the following equations: Based on the calculated RCF Threshold we compare the values based on the input data and mark those over the threshold level as an anomalies. Once anomaly is detected notification process starts.

Once we have everything calculated and compared, the results are sent to another part of the function where we start sending alerts and notifications. For this we created a specific

```python
    #Split the whole file into lines, in a list
    sum_values_list = file_content.splitlines()

    #Remove the date from each element inside the list
    sum_list(sum_values_list)
    del_date(sum_values_list)

    # i += 1
    #Copy the file to another bucket after we finished working on it
    copy_to_bucket('rcf-sagemaker-testdata','rcf-sagemaker-finished', keys_list[0])

    #Json dump
    json_result =  dump_to_csv(del_date_list)

    # return dump_to_json(json_list)
    return json.loads(json_result)


#Remove date from elements taking comma as a delimeter
def sum_list(list):
    for item in list:
        # csv_list.append(item.split(",")[-1])
        csv_list.append(item)

#Remove date from elements taking comma as a delimeter
def del_date(list):
    for item in list:
        del_date_list.append(item.split(",")[-1])

#A standarized way to copy files from 1 bucket to another.
def copy_to_bucket(bucket_from_name, bucket_to_name, file_name):
    copy_source = {
        'Bucket': bucket_from_name,
        'Key': file_name
    }
    s3_r.Object(bucket_to_name,"Data Finished at {} --> ".format(timestr) + file_name).copy_from(CopySource=bucket_from_name + "/"
    # s3_r.Object(bucket_to_name, file_name).copy(copy_source)
    s3_r.Object(bucket_from_name,file_name).delete()
```

Figure 2.25: The moving of data from testing buckets to finished testing data

Slack App & another function that handles all the notifications. This can be found in a detailed section down below under Notifications Functions.

The graph generation was a bit of a complicated manner, since we can't send direct images from lambda to slack without doing a workaround. We had to store the results in a memory buffer, and read that buffer to send it. Notifications being sent had the anomalies, a graph with the data + threshold + anomalies detected in a detailed form.

Using panda, we saved data points in an X (**Date Stamp**) and Y (**Scores**) lists after reading the file from lambda's tmp directory and generate the plot. Then sent to slack with a special

```python
#Generates json files and sends them to a lambda function containing another model call
def dump_to_csv(list):

    #change region_name as per the destination lambda function region
    invokeLam = boto3.client("lambda", region_name="us-east-1")

    data = "\n".join(list)

    #Write to the list to a csv file and store it inside a bucket
    file_name = 'csv_dumps_{}.csv'.format(timestr)
    lambda_path = "/tmp/" + file_name
    s3_path = "csv_dumps/" + file_name
    s3 = boto3.resource("s3")
    s3.Bucket('rcf-sagemaker-finished').put_object(Key=s3_path, Body=data)

    #This will invoke @Nursulta's function, RCF Anomaly Detection.
    resp = invokeLam.invoke(FunctionName = "RCF_Anomaly_Model_2", InvocationType = "RequestResponse", Payload = parse_to_json(csv_l
    # print(resp["Payload"].read().decode())

    # print ("\n\nAll Sum Data from CSV function:\n", data)
    return(resp["Payload"].read().decode())

#Request to model needs to be a json, so this function with parse the data to json before sending it
def parse_to_json(objects):

    # objects = [float(i) for i in objects]
    data = {}
    data["data"] = objects
    json_data = json.dumps(data)
    # print (json_data)
    return json_data
```

Figure 2.26: Invoking RCF-Anomly-Model-2.py

`payload\_2 param`, that holds our channel name, title and some other needed variables for better readability. In general, this all happens in an automatic form.

2.6 Notification functions - Jacek

Notification part is a very important part of our project. After collecting the data, preparing them and using machine learning algorithms, we need to somehow communicate with the user. In this paragraph we will describe the motivation and ways of informing the user about the anomalies and predictions.

```python
#Calculate cut-off final score
scores_array = numpy.array(score_array)
score_mean = scores_array.mean()
score_std = scores_array.std()
score_cutoff = score_mean + 3*score_std

cut_off_final_score=('Cut-Off final Score:',score_cutoff)
```

Figure 2.27: Cut-off score calculator

```python
#Comparing the final cut-off scores with a list of data, these are the final cut-off scores
def compare_cut_off(response_here, list_here):
    print("Final Cut-Off Scores with TIMESTAMP + Score: \n")
    i = 0
    string_2 = ""
    for items in list_here['data']:
        if float(score_array[i]) > response_here:
            # print(str(items.split(",")[0]), float(score_array[i]))
            string_2 = "---".join([str(items.split(",")[0]), str(score_array[i])])
        i += 1
    return ("Anomalies found --> TIMESTAMP + Score:",string_2)

#Used for debugging purposes as well. It prints the received payload to check for errors in logging
def print_payload(payload):
    # print(payload)
    for item in payload['data']:
        print(str(item.split(",")[0]), float(item.split(",")[1]))
```

Figure 2.28: Cut-off comparison

2.6.1 Motivation

To notify the user in order to react for the main events such as anomaly detection, we created a notification functions. Firstly we are calling user by triggering ChatBot API and secondly we are sending notification to the slack channel with a graph and marked anomaly to let user faster react for it.

User story

We decided to create a user story where user who will be notified in case of an alert is one

```python
#Plotting the graph and sending it to Slack Channel
def plot_send(dataFile, anomaly_score, array_results):

    x=[]
    y=[]
    i=0
    data = json.dumps(dataFile['data'])

    #Read the json as a csv
    df=pd.read_json(data)
    df.to_csv('/tmp/results.csv', index=False, header = 0)

    #Grab the csv from tmp in lambda then parse the data into x and y array points
    with open('/tmp/results.csv', 'r') as csvFile:
        reader = csv.reader(csvFile, delimiter=',')
        for row in reader:
            # print((row[0].split(',')[0]))
            #Apped time to X
            x.append(datetime.strptime((row[0].split(',')[0]), '%Y-%m-%d %H:%M:%S'))
            #Append data point to Y
            y.append(float((row[0].split(',')[1])))
            #if point is anomaly, then plot it
            if array_results[i] > anomaly_score:
                plt.plot([datetime.strptime((row[0].split(',')[0]), '%Y-%m-%d %H:%M:%S')], [array_results[i]], marker
            i += 1

    plt.plot(x,y)
    #Horizantal line for Anomaly
    plt.axhline(y=anomaly_score, color='r', linestyle='-', label='Threshold')

    plt.title('Data from CSV: TimepStamps and Scores')

    plt.xlabel('TimeStamps')
    plt.ylabel('Scores')
```

Figure 2.29: Plotting the anomaly graph

of the administrators Bob. Bob and his team are using slack in their work as we do in this project.

In case of the anomaly detection we are triggering ChatBot API to call Bob and notify him about the anomaly. To let Bob faster make his decision and in case of emergency, escalate the problem(or in case of false detection skip it), we decided to show to him the graph where he can visually asses the problem. Since Bob and his team are using slack we decided to send them notification to special slack channel as well.

To accelerate the decision making process, even before logging in to the consoles, cloud watches etc. we decided to send him a part of the graph with prediction and marked anomaly. This will be straightforward information for the Bob or his colleagues what is happening.

The example of notification massages send to the slack channel is depicted in a fig. 2.30.

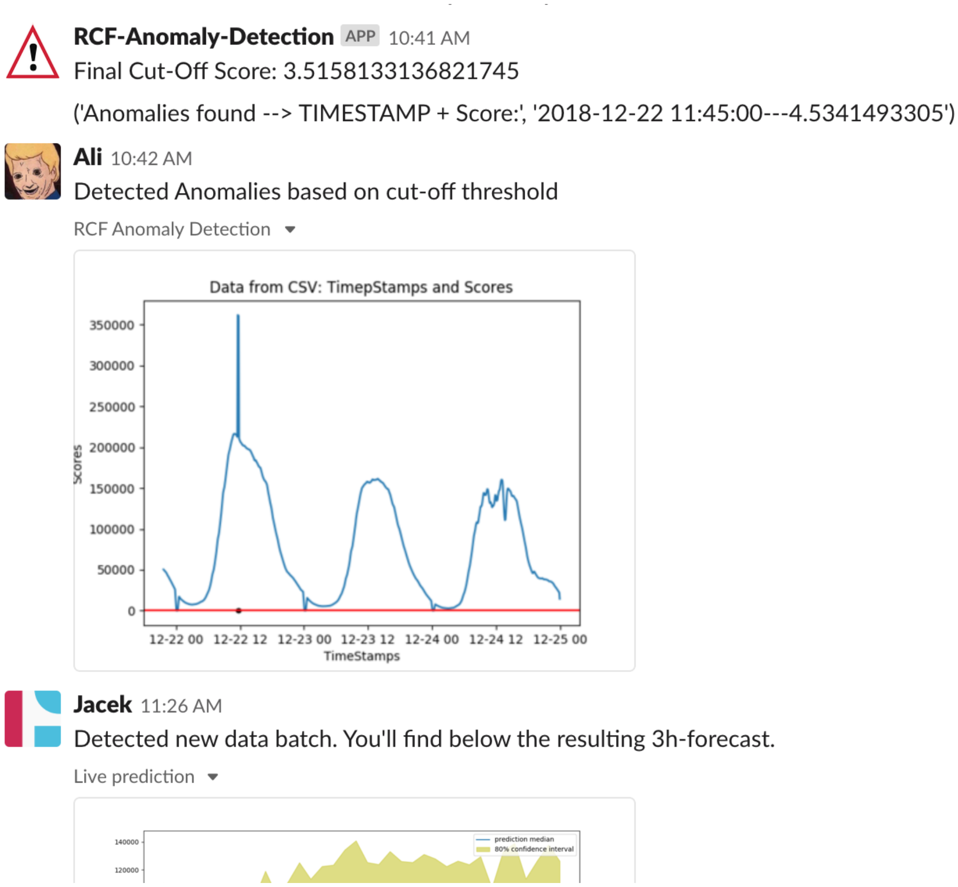


Figure 2.30: Example of anomaly notification

2.6.2 Methods to notify the user about detected anomalies

- **ChatBot API notification** - To notify the ChatBot API in case of an anomaly we are sending REST POST message to provided by ChatBot team URL as it is shown in an example in fig. 2.31. That POST is triggering ChatBot which calls the user.

- **Slack channel metrics** - To send the message with exact metrics and data we are sending the message to the slack channel using Amazon Simple Notification Service topic.

```
function notifyChatBot(url, message) {
  const headers = {
    'Content-Type': 'application/json',
    'Authorization': 'Bearer xyz'
  }
  const request = {
          status: 'firing',
          description: message,
          priority: 'high'
        }

  console.log("This post will call our friends at ChatBoT: %j", request);
  axios.post(url, request, {headers: headers})
  .then(function (response) {
    console.log('Alles Gut!');
    console.log(response);
  })
  .catch(function (error) {
    console.log('Oh no something went wrong!');
    console.log(error);
  });
```

Figure 2.31: Example of ChatBot API notification

```
#Send Slack MSG
sns = boto3.resource('sns')
topic = sns.Topic('arn:aws:sns:us-east-1:746022503515:RCF-SageMaker')

response = topic.publish(
    Message=str("Final Cut-Off Score: {}".format(score_cutoff)),
    Subject='#aws',
    MessageStructure='text/plain',
)
```

Listing 2.4: Sending message to slack channel using SNS topic

- **Slack channel graph** - Unfortunately to send the graph to the slack channel method with using SNS topic was not working correctly. For showing the graph using AWS lambda function we needed to do the following steps:

  1. Plot a new graph in memory.
  2. Save that graph to a buffer and go to the beginning of a buffer

```
buf = io.BytesIO()
plt.savefig(buf, format='png')
buf.seek(0)
```

3. Add a buffer to a structure

```
my_file = {
'file' : ('./buf.jpg', buf, 'png')}
```

4. Add that structure to the REST POST and post that file to a slack channel.

```
r = requests.post("https://slack.com/api/files.upload",
    params=payload, files=my_file)
```

2.7 Project delivery - Alex

After six months of intensive work, we ended up with a pretty diverse code base that we needed to unify. We had various models, lambda functions, EC2 instances, roles and policies composing our processing pipeline, and hand them in the most convenient way.

2.7.1 Terraform

The source code for this part is located in `terraform/`.

We chose to use the infrastructure-as-code deployment tool Terraform, with a few quirks that are presented below.

Terraform uses a `state` to keep track of the state of managed cloud components, and perform its changes. This file is created locally on the machine running the terraform commands, but problems arise when two or more people try to concurrently run terraform scripts at the same time : inconstistencies could arise from such situations, which is to avoid at all costs.

To deal with this problem, we added the following piece of code:

```

terraform {
 backend "s3" {
   bucket         = "fog-bigdata-terraform-backend"
   dynamodb_table = "terraform-state-lock-dynamo"
   key            = "backend/terraform.tfstate"
   region         = "${var.region}"
   encrypt = true
  }
}
```

Which hosts this `state` in a remote s3 bucket, allowing thanks to a locking mechanisme several people to work concurrently on the terraform files.

2.7.2 Lambda functions deployment

AWS has recently released a lambda function management tool called SAM, but unfortunately Terraform does not support it yet, so we had to package our lambda functions in ZIP files (ref) and use Terraform's `aws_lambda_function` to deploy them.

We had to overcome some hurdles, which are built-in AWS limitations regarding lambda functions:

- Zipped deployment package size limited to 50MB
- Unzipped deployment package, including layers, limited to 250MB

Since we are using the Matplotlib and Pandas librairies, we ended up with deployment packages weighting more than the AWS limit.
That's why we came up with the idea of creating our own layers (thanks to `aws_lambda_layer_version`), and linking our lambda functions to those layers. Thus, we end up with ligthweight lambda functions (1MB), and also reusable layers.

2.7.3 IAM roles and policies

Every AWS service needs the correct policies to access the other components it works with. For instance, our lambda functions need to access AWS S3 and AWS SageMaker, and this is fortunately well supported by terraform.
The code below creates:

- A role that can only be assumed by AWS Lambda services

- Two policies giving full access to S3 and Sagemaker, attached to this previous role.

```
resource "aws_iam_role" "lambda_sm_s3_role" {
  description="Role giving lambda full access to S3 and SageMaker"
  assume_role_policy = <<EOF
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "",
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
EOF
}

data "aws_iam_policy" "AmazonS3FullAccess" {
  arn = "arn:aws:iam::aws:policy/AmazonS3FullAccess"
}

data "aws_iam_policy" "AmazonSageMakerFullAccess" {
  arn = "arn:aws:iam::aws:policy/AmazonSageMakerFullAccess"
}

resource "aws_iam_role_policy_attachment" "s3-full-access" {
  role       = "${aws_iam_role.lambda_sm_s3_role.name}"
  policy_arn = "${data.aws_iam_policy.AmazonS3FullAccess.arn}"
}

resource "aws_iam_role_policy_attachment" "sm-full-access" {
  role       = "${aws_iam_role.lambda_sm_s3_role.name}"
  policy_arn = "${data.aws_iam_policy.AmazonSageMakerFullAccess.arn}"
}
```

2.7.4 EC2 Instances

As previously mentioned in 2.1.3, we need to spin up an EC2 instance, but also upload and run a script generating a fake stream of data to feed our processing pipeline.

Since Terraform is only an infrastructure management tool, we needed an additional provisioning tool for this task. We went with Ansible, since it's pretty straightforward to use.

Thus, terraform calls a specific command after the creation of the EC2 instance, which runs an Ansible playbook (in `utils/ec2-provisioning.yml` charged with running the script.

Ansible provisioning

Since Terraform allows us to run specific scripts after provisioning of a resource, we used this hook to trigger an Ansible job dealing with the provisioning of our EC2 instance, and a few points are worth mentioning:

- Since we could want to SSH to the EC2 instance, we also used Terraform to manage an `aws_key_pair`, which is the public part of an SSH key pair of our choice. The private part is hosted on the machine running the Terraform script, and of course not commited to the source code.
- When considering the followinng piece of code :

```
        provisioner "local-exec" {
            command = "rm ansible-hosts.ini &&
            echo \"${self.public_ip} ansible_user=ec2-user\" >> ansible-hosts.ini
            && ansible-playbook ec2-provisioning.yml --private-key ~/Downloads/default-vpc-access.pem -i ansible-hosts.ini"
        }
```

  We see that we have to first append to `ansible-hosts.ini` the public IP of our EC2 instance. This is because Ansible needs to know the IPs of its managed servers, in order to provision them.

  Also, the `--private-key` argument should be changed to point to the location of the private key used to SSH to the server.

2.7.5 S3 Notifications & SNS Topics

Our lambda functions are triggered by SNS Topic events, monitoring uploads on specific endpoints of S3. We need to create those topics, and also the S3 notifications that send events to those, in order to have a working pipeline.

This is handled by Terraform, with the `aws_s3_bucket_notification` and `aws_sns_topic` directives, that take care of spinning up those services with the correct roles and policies.

2.7.6 S3 Buckets

The management of S3 bucket with Terraform can be tricky, since calling `terraform destroy` will delete all existing buckets and get rid of all our data! That's why we didn't import the bucket holding all our data to Terraform.

Instead, we create a new bucket called `sanitized-datasets`, which will after creation be populated by a script (`data-preprocessing.py`) with subfolders, each containing our data correctly formatted for each algorithm. For instance, `s3://sanitized-bucket/deep_ar/` will contain data with a DeepAR-compatible shape. This is implemented by using the `provisioner` parameter, which allows us to run local provisioning scripts after the creation of a resource. In this case, the provisioner takes care of populating the `sanitized-datasets` bucket with correctly-shaped data.

2.7.7 What was not covered

Even though pretty much everything is managed by Terraform, because of the high velocity of AWS development, some features were missing.

- Our lambda functions are usually triggered by SNS Topic events, but unfortunately this is not managed yet in Terraform. As a consequence, one has to add those triggers manually (on the AWS Lambda web editor).
- Also, SageMaker support is almost inexistent. That's why we had to leverage Teraform's `Provisioners` in order to train & deploy our models. If you consider the following code:

```
    // Train a MeanPredictor model and export it as endpoint
```

```
    provisioner "local-exec" {
      command = "python ../models/mean_predictor/train_deploy.py --trainpath s3://${aws_s3_bucket.datasets.bucket}/rcf/data/train/data.csv --role ${aws_iam_role.sm_role.arn} --freq ${var.data_aggregation_frequency}"
    }

    // Train a DeepAR model and export it as endpoint
    // WARING: takes ~ 2 hours
    provisioner "local-exec" {
      command = "python ../models/deep_ar/train_deploy.py"

      environment {
        BMW_DATA_BUCKET       = "fog-bigdata-bmw-data"
        SANITIZED_DATA_BUCKET = "${self.bucket}"
        SAGEMAKER_ROLE_ARN    = "${aws_iam_role.sm_role.arn}"
        ENDPOINT_NAME         = "${var.deepar_endpoint_name}"
        DATA_FREQUENCY        = "${var.data_aggregation_frequency}"
      }
    }
```

The `local-exec` parts are the lines taking care of training and deploying our models. They are executed after provisioning of the S3 bucket holding all the correctly-shaped data.

Chapter 3

Project management techniques - Marina

This chapter will give an overview to the ways this project was managed internally, as well as by the supervisors. It includes a detailed introduction to the Kanban - management tool, followed by complementary techniques used by the team to further improve the working process.

3.1 Kanban

For internal management of the team, based on the suggestions of our industry partners, it was decided to use an Agile product management technique - Kanban. It emphasizes on a continuous delivery mindset, trying not to overburden the development team. Like other Agile techniques, it was designed to facilitate communication and division of work inside the team, while giving live updates so each member can keep up with the process. The simple and intuitive structure helps the team work in a more efficient way.

3.1.1 Principles

Kanban is build by respecting the following 3 principles:

1. **Visualization**: Kanban uses mechanisms organised in a Kanban board. The board, by presenting all the tasks at once, in an easy to grasp manner, gives the developer a better understanding of the context, thus facilitating the workflow.

2. **Limited amount of work**: because the number of tasks are limited from a week to another (as well as the number of cards stored in production) the team does not feel overburdened with work, and is not pressured to deliver something by sacrificing on the quality aspect.

3. **Continuous flow**: when a member is done with his/her tasks, he/she can just move forward with the work process by picking up the next tasks (placed on the top of the stack). This way nobody sits around waiting to be designated with work, thus ensuring a continuous workflow.

3.1.2 Benefits

Taking in consideration the above mentioned information, it is easy to identify the way this project management method can improve the delivery process of a product, following are a couple of its advantages:

1. *Kanban methodology utilizes short working cycles.* In the case of this project the cycle was limited at one week (with an exception during the winter break). This 1 week cycles ensured a continuous delivery process, where the industry partners had the opportunity to keep the team in check, by introducing new features/corrections in the process.

2. *Kanban is very responsive to change and feedback.* As mentioned above, thanks to a short cycle, the BMW representatives had the opportunity to introduce change in an

utile time, by coming up with new features or scenarios, as well as feedback to improve or correct the direction of the project.

3. *Kanban removes time wasting activities.* While popular management techniques require a series of mandatory daily meetings and discussions, Kanban eliminates this type of time wasting activities. Meetings were organized once a week or on team's request, in cases of confusion or identification of impediments of the working process that should be discussed and solved at team level.

3.1.3 Roles

Given the Kanban approach, it does not have well predefined roles. It suggests to start from the structure of your team, and current management, and to evolve it as seen necessary, taken in consideration the specific of the methodology. As the team had no previous experience working together, it decided to have a Project Manager that would facilitate the communication between them and the supervisors, as well as take care of organizational issues. The project manager's responsibilities are:

- Managing the Kanban board (in Trello) which means writing down the tasks, assigning responsible people, adding labels, deadlines (if necessary), updating the progress, and cleaning up the board by periodically archiving the old cards.
- Managing other means of communication, like Slack, by organizing specific channels, setting up notifications and reminders.
- Setting up and managing meetings, making sure they are productive and that the team is up to date with project's progress.
- Being the spokesperson for the team, mediating the communication with the supervisors.

3.1.4 Kanban board

As already mentioned above, Kanban uses a board as it's main management tool. A Kanban board is designed to help visualise the workflow. A typical board can be broken down into the following elements:

1. Cards or Visual Signals
2. Columns
3. Work in Progress (WIP) Limits

The visual signals are usually the tasks that are written on stickies, in the case of a physical board, or on cards, in the case of software based boards. Each card encapsulates one user story, and all the user stories should require an equal or at least close enough effort. Once written down and attached to the board, these visual signals help the team, as well as other stakeholders, understand the product development. Columns represent different production stages that an user story can be in. All these stages together compose the workflow. The columns can be as simple as “To do”, “In progress” and “Done”, to more complicated stages, depending on the complexity of the development process. Work in Progress Limits are the maximum number of cards that can be placed in a certain column at a certain period of time. This is done in order to limit the team's ability to commit to too much work in a short period of time.

Trello as a Kanban board

For the project, the team decided, following the supervisors' suggestion, to use a digital board. This allows it not to share a physical space (like an office), and to work remotely and asynchronously. Trello is one of the simplest and easiest way to mimic a Kanban board by allowing the users to create lists, that mimic the columns, and cards that represent user stories.

For this project, it was decided to create the following workflow structure:

- Backlog - aggregates all user stories
- Breakdown (Doing and Done)- splits up user stories in smaller units of work
- Implementation (Doing and Done) - indicates what cards are currently in work
- Validation (Doing and Done)- tests the finished user stories against the definition-of-done.

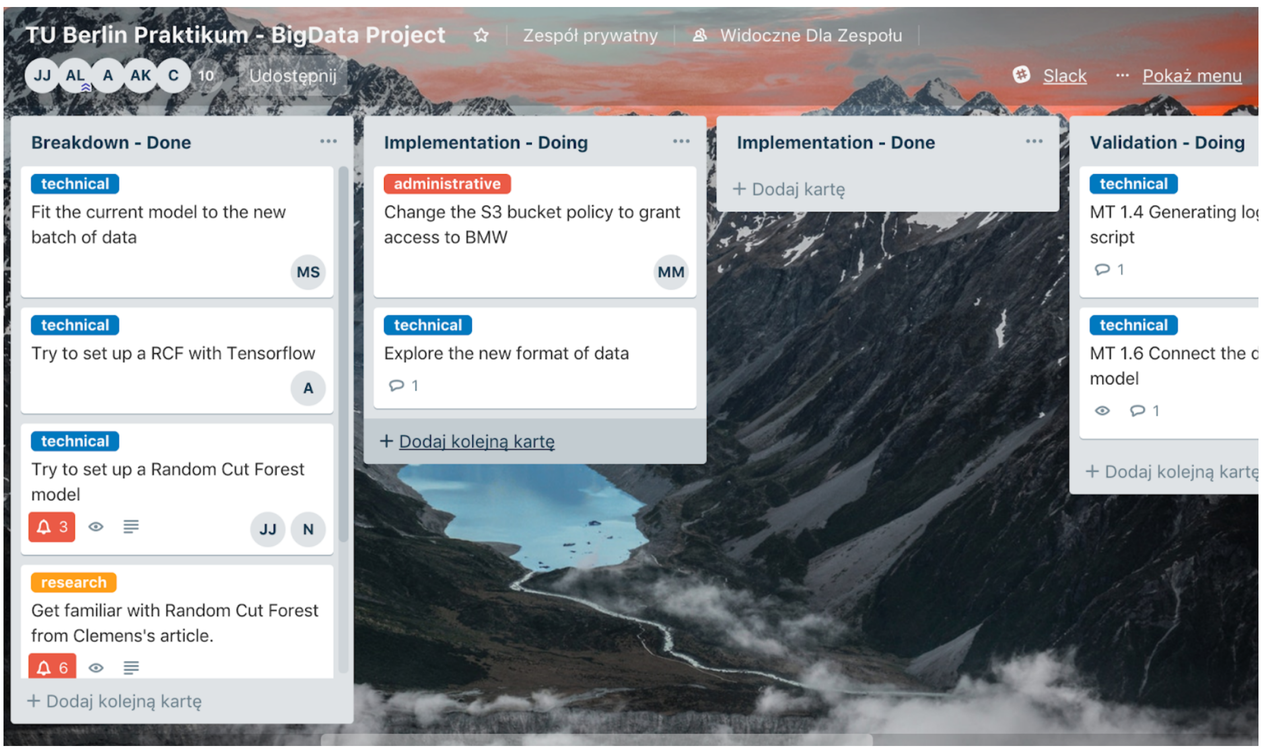


Figure 3.1: Big Data Analytics Trello board

After setting up the above mentioned columns, a user can start adding cards/user stories to the backlog. As seen in figure 3.1, Trello allows an user to add a title, a description (to detail the card), attachments, comments, due dates, responsible members and labels to a card. It's worth noting, that for this project, the team used the board not just for programming related tasks. The Trello board was used for managing all project related assignments, for the purpose of registering each week's work. In this project, the team used the following

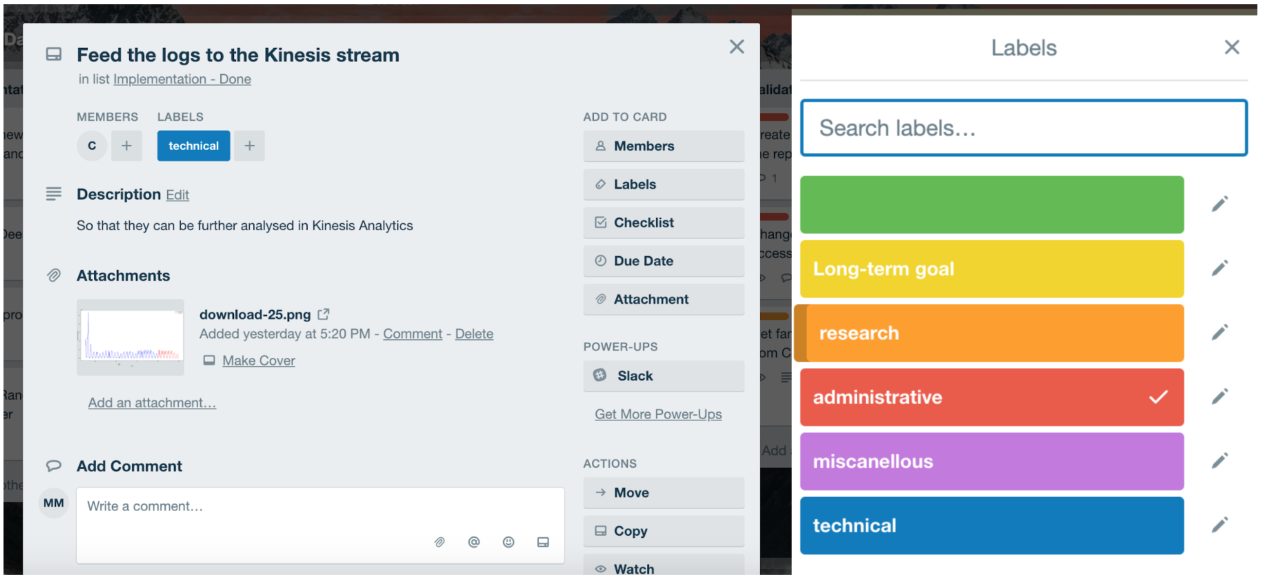


Figure 3.2: Trello board card view with labeling details

series of labels to mark the purposes of the user-story (see **??**):

- **Long-term goal**: indicates that the card is most likely a use case in the project, which should be broken down further into smaller tasks
- **Research**: marks tasks that require the team to look for papers, documentation, solutions to solve certain impediments
- **Administrative**: highlights that a task has a managerial character
- **Technical**: indicates a developer task;
- **Miscellaneous**: used for any other subjects that a task can have, except the ones mentioned above (e.g. creating the presentation or documentation paper)

3.2 Meetings

3.2.1 Internal meetings

The team organized internal weekly meetings to discuss features, plan and assign the tasks for the following production cycle. The meeting was limited at a 2 hour mark in order to keep it productive and effective and remove time wasteful discussions. The time frame in which the meeting was set, was agreed by all team members based on each individual schedule, keeping in mind that is should offer enough working horizon before the next meeting with the supervisors. Internal meetings were structured in the following way:

1. Discussing feedback from the last meeting with the supervisors
2. Adjusting the Kanban board with new tasks, or modifying the existing tasks with new requirements or details
3. Defining the goal of the following sprint/week
4. Picking up new tasks, discussing the details of each tasks, requirements and impediments
5. Assigning tasks to each team member, depending on the complexity and each member's experience with a certain technology, sometimes by forming smaller sub-teams
6. Aggregating a list of questions for the next meeting with supervisors
7. Summarizing the meeting in a 2-3 minutes recap version.

3.2.2 Meetings with supervisors

To keep the process reactive and to assure that the product is in a continuous development, meetings were organized with supervisors from the TUB, BMW and AWS. The meetings followed a cyclic pattern, being organized once a week and lasting from 30 to 60 minutes (based on each party's availability). The structure of the meeting was also repetitive, building in the following way:

1. *Progress update.* The development team starts each meeting with presenting the current progress of the project, usually by demo-ing some part of the code, other times by explaining the research and investigations done in the previous week.
2. *Q and A session.* After demo, the development teams present the impediments they identified during the working process, pointing out to some issues or asking for help or assistance is solving them. Also, any uncertainties are discussed and clarified.
3. *Feedback round.* The industrial partners evaluate the progress by commenting on the current state of the project, pointing to things that can be improved. They discuss potential new features, or directions the project can go.
4. *Planning round.* Based on the received feedback, the development team comes with a proposition on things that can be done in the following cycle.

Chapter 4

Conclusions and Recommendations

Taking in consideration all the approaches mentioned throughout the paper for both prediction and anomaly detection models, and the fact that for BMW this is a more recent project, meaning that it does not possess a great amount of data to train and validate models, here are the project recommendations:

1. Start the framework by implementing a *Mean Predictor* algorithm for both anomaly detection and prediction. It trains well for small amount of data, takes less computational power, and will offer reliable results.
2. Gather data for at least a year and save all the data in different granularity buckets. Plot and test the data and see what granularity offers best results compared to the computational power needed to process it. In terms of this project, 5 minutes granularity buckets proved to perform good, preserving all the anomalies while requiring less computational power to pre-process.
3. Add a *geolocation* component to the data and create different data pools for each regions of the globe, instead of smashing all information into one bucket. Such a change might give new insights into the gathered information. For example, it might reveal different holiday patterns for different countries, as well as take into consideration the specific

holidays for different regions (christian celebration vs. muslim celebrations vs. asian holidays).

4. After having enough data, eventually even data that includes a location component too, run it through a *DeepAR* model. As this algorithm allows the analysis of a couple of sources of data at the same time, try including other sources of data like weather or gas prices. This algorithm (for a significant amount of data) can find some dependencies between different sources of data and improve the predictions.

5. In a year or two, implement a *Random Cut Forest* model. After gathering some expert knowledge in handling anomalies, as well as information about the cause behind the anomalies and which deviations are worth looking into, you will have a better intuition about the right threshold. It might be an interesting idea to set a couple of threshold for different levels of anomalies, if this actually proves to bring any advantage in the way the anomalies are managed down the line.

6. Run the data (after at least a year) through a *Holt-Winters* model and try to identify seasonal trends (if they actually exist). Verify if the trends depend of the season/period of the year, some holidays or the geographical region. Such a model should offer new interesting insight into your data.

Appendices

Appendix A

Real-time Anomaly Detection in VPC Flow Logs (in AWS)

A.1 Introduction

Credit goes to Igor Kantor (https://medium.com/@devfire) who wrote the original post (5 parts) on Medium:

The goal of this GitHubGist is to support anyone who wants to implement the described architecture and get it running on AWS. This means you should use both the Medium Post and this GitHubGist for the implementation (since I will not repeat all the text here).

On my aws account I used a prefix (medium) for all services, to easily find them amongst all the other running services/instance/funtions/roles etc. (just as a suggestion). It will make cleaning up your aws account easier later on.

In this GitHubGist we will focus on the anomaly detection ("components within the red dashes"):

A.2 Part 1

https://medium.com/@devfire/real-time-anomaly-detection-in-vpc-flow-logs-part-1-introduction-55ed000e039b

Just introduction text. Does not contain any code or implementation.

A.3 Part 2

https://medium.com/@devfire/real-time-anomaly-detection-in-vpc-flow-logs-part-2-proposed-architecture-32683755abf7

Gives an overview of the architecture.

In this GitHubGist we will focus on the anomaly detection ("components within the red dashes").

A.4 Part 3

https://medium.com/@devfire/real-time-anomaly-detection-in-vpc-flow-logs-part-3-kinesis-stream-1bdd8a9426f1

This is where the fun begins.

Make sure you have the AWS Command Line Interface installed (https://docs.aws.amazon.com/de_de/cli/latest/userguide/cli-install-macos.html).

If you just installed the aws cli, make sure you run

```
aws configure
```

to connect it to your aws account.

Let's get started (with the article):

```
aws kinesis create-stream --stream-name "VPCFlowLogs" --shard-count 1
```

```
vim allowCloudWatchAccesstoKinesis.json
```

Paste the content and substitute your region as needed (in line: "Principal": "Service": "logs.**us-east-1**.amazonaws.com" ,).

Run the following command **from the same directory** where **allowCloudWatchAccesstoKinesis.json** is located:

```
aws iam create-role --role-name CloudWatchToKinesisRole --assume-role-policy-document file://./allowCloudWatchAccesstoKinesis.json
```

```
vim cloudWatchPermissions.json
```

Paste the content and **substitute the two resource arn strings** with yours.

Run the following command **from the same directory** where **cloudWatchPermissions.json** is located:

```
aws iam put-role-policy --role-name CloudWatchToKinesisRole --policy-name Permissions-Policy-For-CWL --policy-document file://./cloudWatchPermissions.json
```

```
aws logs put-subscription-filter \
    --log-group-name "VPCFlowLogs" \
```

```
    --filter-name "VPCFlowLogsAllFilter" \
    --filter-pattern "[version,␣account_id,␣interface_id,␣srcaddr␣!=␣"
        -",␣dstaddr␣!=␣"-",␣srcport␣!=␣"-",␣dstport␣!=␣"-",␣protocol,␣
        packets,␣bytes,␣start,␣end,␣action,␣log_status]" \
    --destination-arn "arn:aws:kinesis:us-east-1:31415926:stream/
        VPCFlowLogs" \
    --role-arn "arn:aws:iam::31415926:role/CloudWatchToKinesisRole"
```

```
brew install jq

aws kinesis get-records --limit 10 --shard-iterator $(aws kinesis get-
    shard-iterator --stream-name medium_VPCFlowLogs --shard-id shardId
    -000000000000 --shard-iterator-type TRIM_HORIZON | jq -r ."
    ShardIterator") | jq -r .Records[].Data | base64 -d | zcat
```

In my case

```
aws kinesis get-records --limit 10 --shard-iterator $(aws kinesis get-
    shard-iterator --stream-name medium_VPCFlowLogs --shard-id shardId
    -000000000000 --shard-iterator-type TRIM_HORIZON | jq -r ."
    ShardIterator")
```

gives me the following:


```
{
    "Records": [],
    "NextShardIterator": "AAAAAAAAAAF3vrrgukNNTm4aHO8iRO5DXESr+
        ofQJFH57eXltrr1DTJW1DExKKckOxU5ZT2nFoQ2FOJsmfscXDRl9po0Q9Jb1Zvs9aytKcMLldmE6P7o
        /HZSC5uhlpT3/tFVAAMQAvqZ41MIFJqFik/
        kShb9q9oEB3WbYTrigihimcr1haixhIdQv2J6TJhJ6dZ1l6ggsVcnjbok1NZzIyzeABkS8fhLKSsJ/
        qIV+WmNigYX0MSYVA==",
    "MillisBehindLatest": 31385000
}
```


so when appending the

```
| jq -r .Records[].Data | base64 -d | zcat
```

part to that command it gives me "no matches found: .Records[].Data". So I still have to figure out how to insert records here (probably just need to put traffic on the vpc).

A.5 Part 4

https://medium.com/@devfire/real-time-anomaly-detection-in-vpc-flow-logs-part-4-kinesis-analytics-c80cc4977e97

Navigate to the Kinesis Analytics page in the AWS console and click on Create Application.

Name it VPCFlowLogsAnalytics.

Hook it up to the VPCFlowLogs Kinesis stream you created earlier.

in case you can not create the application because you don't have enough

*data see section **Amazon Kinesis Data Generator** on the bottom of this file.*

Enable Lambda pre-processing and say you want a new Lambda function.

Use blueprint **kinesis-analytics-process-compressed-record** for the lambda function

Name the lambda function *KinesisAnalyticsProcessCompressedRecord*

Create in IAM a **lambda_kinesis_exec_role** and give it AmazonKinesisFullAccess.

Assign **lambda_kinesis_exec_role** for your lambda function (*KinesisAnalyticsProcessCompressedRecord*).

Increase the Timeout setting to at least 1 minute

Click on Discover Schema

In my case the schema was not recognized and I had to manually create all the columns:

I had to rename the columns **start** and **end** (see error message in screenshot). Here are the column names:

```
version, account_id, interface_id, srcaddr, dstaddr, srcport, dstport,
    protocol, packets, bytes, start_, end_, action, log_status
```

I recommend to create them in reverse order (log_status first, version last) so you dont have to sort manually.

A.6 Part 5

https://medium.com/@devfire/real-time-anomaly-detection-in-vpc-flow-logs-part-4-kinesis-analytics-c80cc4977e97

We are still where we left in part 4 (create Kinesis Analytics Application):

Click on the blue **Go to SQL Editor** button.

```
-- \textbf{ Anomaly detection \textbf{
-- Compute an anomaly score for each record in the source stream using Random Cut Forest
-- Creates a temporary stream and defines a schema
CREATE OR REPLACE STREAM "TEMP_STREAM" (
--   "APPROXIMATE_ARRIVAL_TIME"     timestamp,
--   "srcaddr"      varchar(16),
--   "dstaddr"    varchar(16),
   "bytes"        DOUBLE,
   "ANOMALY_SCORE"  DOUBLE,
   "ANOMALY_EXPLANATION"  varchar(512));

-- Creates an output stream and defines a schema
CREATE OR REPLACE STREAM "DESTINATION_SQL_STREAM" (
--   "APPROXIMATE_ARRIVAL_TIME"     timestamp,
 --  "srcaddr"      varchar(16),
--   "dstaddr"    varchar(16),
   "bytes"        DOUBLE,
   "ANOMALY_SCORE"  DOUBLE,
   "ANOMALY_EXPLANATION"  varchar(512));

-- Compute an anomaly score for each record in the source stream
-- using Random Cut Forest
CREATE OR REPLACE PUMP "STREAM_PUMP" AS INSERT INTO "TEMP_STREAM"
```

```
-- SELECT STREAM "APPROXIMATE_ARRIVAL_TIME", "srcaddr", "dstaddr", "bytes", "ANOMALY_SCORE", "ANOMALY_EXPLANATION"
-- SELECT STREAM "srcaddr", "dstaddr", "bytes", "ANOMALY_SCORE", "ANOMALY_EXPLANATION"
  SELECT STREAM "bytes", "ANOMALY_SCORE", "ANOMALY_EXPLANATION"
  FROM TABLE(RANDOM_CUT_FOREST_WITH_EXPLANATION(
    CURSOR(SELECT STREAM "bytes" FROM "SOURCE_SQL_STREAM_001"), 100, 256, 100000, 1, true
  )
);
-- Sort records by descending anomaly score, insert into output stream
CREATE OR REPLACE PUMP "OUTPUT_PUMP" AS INSERT INTO "DESTINATION_SQL_STREAM"
SELECT STREAM * FROM "TEMP_STREAM"
--WHERE ANOMALY_SCORE > 3.0
ORDER BY FLOOR("TEMP_STREAM".ROWTIME TO SECOND), ANOMALY_SCORE DESC;
```

(It is mentioned that credits go to Curtis Mitchell: https://medium.com/@curtis_mitchell)

To understand where the anomaly detection happens see:

https://docs.aws.amazon.com/kinesisanalytics/latest/sqlref/sqlrf-random-cut-forest-with-explanation.html

The following part is not from the Medium article, but was added for clarification

A.7 Part 5 continued

Create new lambda function using the blueprint **kinesis-analytics-output**

Name the lambda function **kinesis_analytics_destination**

Open your *VPCFlowLogsAnalytics* in *Kinesis Analytics applications*

Click **Connect new destination** button

Choose *AWS Lambda function* as destination and select **kinesis_analytics_destination**

As *In-application stream name* choose **DESTINATION_SQL_STREAM**

Click **Save and continue**

A.8 Amazon Kinesis Data Generator (used for part 4)

To supply the Kinesis Stream with data I had to set up an **Amazon Kinesis Data Generator** first.

Open the page https://awslabs.github.io/amazon-kinesis-data-generator/web/help.html.

Download the CloudFormation template https://s3-us-west-2.amazonaws.com/kinesis-helpers/cognito-setup.json

Click the blue **Create a Cognito User with CloudFormation** button (on the https://awslabs.github.io/amazon-kinesis-data-generator/web/help.html).

You are redirected to aws console.

Choose **Upload a template to Amazon S3** and choose the **cognito-setup.json** which you downloaded before.

Click the **next** button (bottom right).

Read the rest of the https://awslabs.github.io/amazon-kinesis-data-generator/web/help.html to learn how to access and use the Amazon Kinesis Data Generator.

A.9 allowCloudWatchAccesstoKinesis.json

```
{
  "Statement": {
    "Effect": "Allow",
    "Principal": { "Service": "logs.us-east-1.amazonaws.com" },
    "Action": "sts:AssumeRole"
  }
}
```

A.10 cloudWatchPermissions.json

```
{
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "kinesis:PutRecord",
      "Resource": "arn:aws:kinesis:us-east-1:31415926:stream/
         VPCFlowLogs"
    },
    {
      "Effect": "Allow",
      "Action": "iam:PassRole",
      "Resource": "arn:aws:iam::31415926:role/CloudWatchToKinesisRole"
    }
  ]
}
```

Bibliography

[1] *AWS - Random Cut Forest with explanation*. URL: https://docs.aws.amazon.com/kinesisanalytics/latest/sqlref/sqlrf-random-cut-forest-with-explanation.html.

[2] *DeepAR Forecasting algorithm*. URL: https://docs.aws.amazon.com/sagemaker/latest/dg/deepar.html.

[3] Charles C. Holt. "Forecasting seasonals and trends by exponentially weighted moving averages". In: *International Journal of Forecasting* 20.1 (2004), pp. 5–10. DOI: https://doi.org/10.1016/j.ijforecast.2003.09.015.

[4] *Netflix Quote*. URL: https://medium.com/@devfire/real-time-anomaly-detection-in-vpc-flow-logs-part-2-proposed-architecture-32683755abf7.

[5] *Real-time Anomaly Detection in VPC Flow Logs*. URL: https://medium.com/@devfire/real-time-anomaly-detection-in-vpc-flow-logs-part-1-introduction-55ed000e039b.